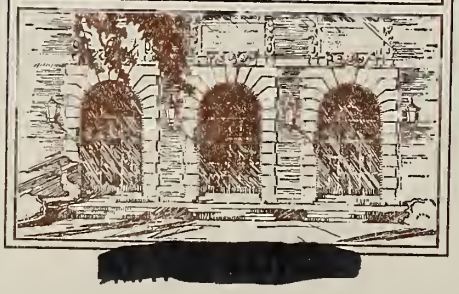


UIUCDCS-R-75-710

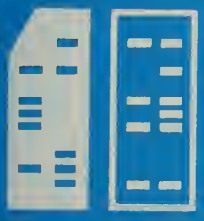
COO-2383-0019

A GRAPHICS INTERPRETER LANGUAGE

BY

JAMES BRENDAN STYNES

May, 1975



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-75-710 COO-2383-0019

A GRAPHICS INTERPRETER LANGUAGE

BY

JAMES BRENDAN STYNES

May, 1975

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

*Supported in part by the Atomic Energy Commission under contract US AEC AT(11-1) 2383 and submitted in partial fulfillment of the requirements of the Graduate College for the degree of Master of Science in Computer Science.

ACKNOWLEDGMENT

I would like to thank Professor C. W. Gear for his supervision and many ideas, Al Whaley for his solution to numerous problems, Neal Hennegan and Carlo Segre for their implementation of the language, and Pamela Farr for her tireless efforts in typing and preparing this thesis.

TABLE OF CONTENTS

1. Introduction. 1

2. Choice of Data Structure. 4

3. Choice of Language and Language Structure 9

3.1 Interpreter Language 9

3.2 Types of Variables 11

3.2.1 Dynamic Variables 13

3.2.1.1 Address Calculation of Dynamic Variables . . 18

3.2.2 Static Variables. 26

4. Instruction Set 28

4.1 Arithmetic and Logical Commands. 30

4.2 Control Commands 39

4.3 Text Buffer Commands 45

4.4 Data Structure Commands. 60

4.4.1 File Commands 66

4.4.2 Block Commands. 67

4.4.3 Entry Commands. 69

4.4.3.1 Information Word and X - Y Quad Commands . . 75

4.4.3.2 Text Group Commands. 78

4.5 Input/Output Commands. 84

4.5.1 Input/Output Control. 84

4.5.2 Console Commands. 94

4.5.2.1 Display Commands 96

4.5.2.2 Grids. 100

4.5.2.3 Input Commands 107

4.5.3 Storage Commands. 111

4.5.3.1 Remote Communication Commands. 111

4.5.3.2 Local Storage Commands 112

4.5.4 Miscellaneous Devices 114

4.6 Subroutines. 115

4.6.1 Parameters. 118

List of References. 122

Appendix. 123

A. Opcode Description. 124

B. Instruction List. 130

C. Subroutine Linkage. 131

1. Introduction

A general purpose simulation and modeling system has been developed at the University of Illinois, enabling a user to represent a model by means of a pictorial description composed of elements, their connection points, and mathematical equations. These describe the functions of the elements and their interrelationships. When the description is complete, the model may then be analyzed, and, finally, the results are presented to the user for examination and interpretation (see [1] and [2]).

A simplified picture of the configuration used in the implementation of this system is depicted in Figure 1. At present, a user may construct his model at any one of four display terminals, each possessing a keyboard, joystick, and display screen. These terminals are connected to a terminal controller, PDP 8/I, which controls the passage of display files to and from the terminals. The controller, in turn, is linked to a PDP/8, which handles the manipulation of the data structures, and maintains a temporary library of user files on the disk. The IBM 360/75 and the PDP/10 are connected to the PDP/8 by a remote transmission link, and are used to maintain permanent file libraries and to perform model analysis. Thus, the hardware configuration pictured in Figure 1 is logically divided into two parts:

1) Local System. All of the hardware pictured above the dotted line enables the user to describe the model, that is, to specify the elements composing the model in terms of their pictorial representation and mathematical equations.

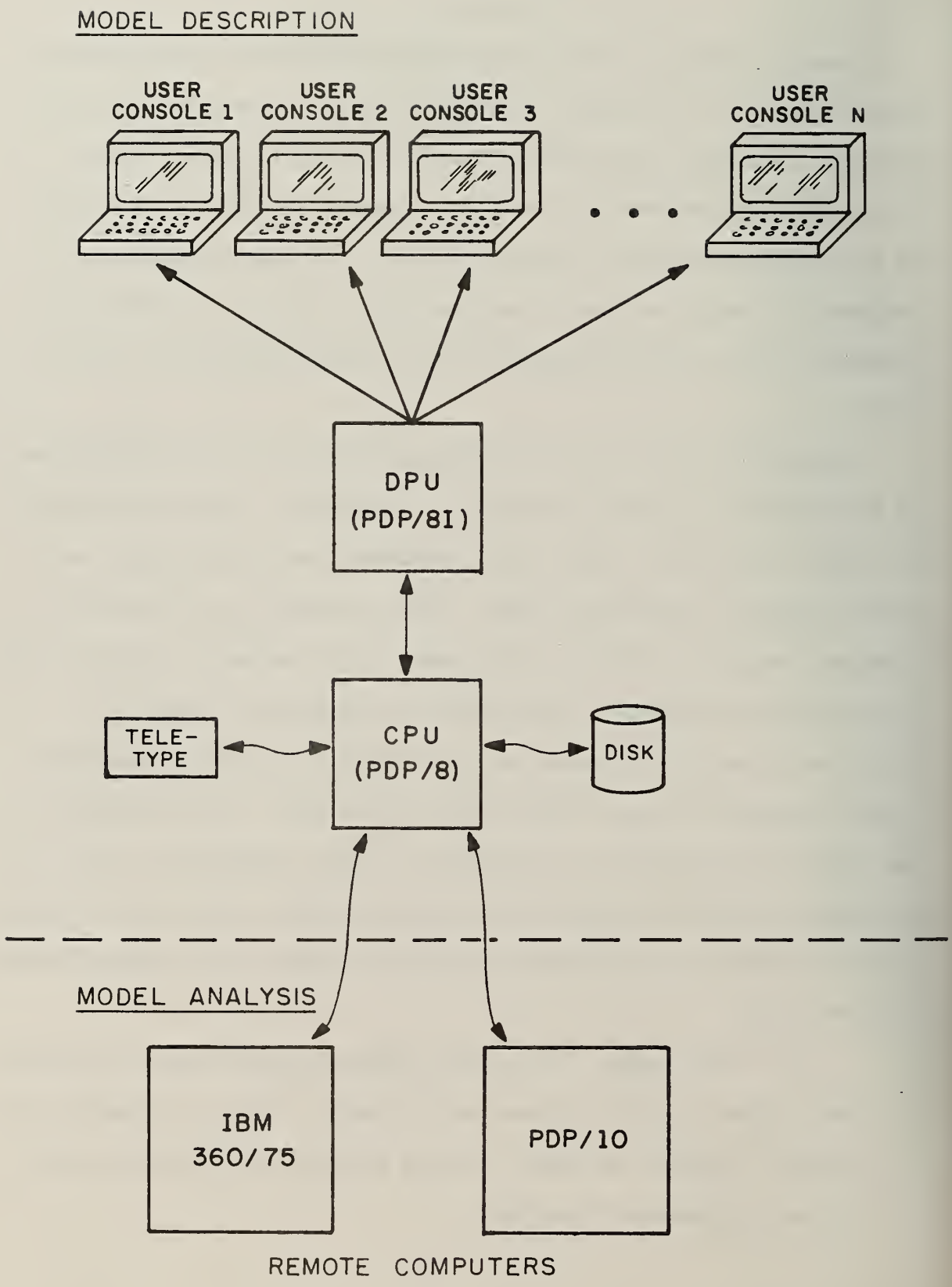


Figure 1 Model Description

2) Remote System. All of the hardware pictured below the dotted line performs the analysis of the model constructed in the local system.

This report is concerned with further facilitating model description. Specifically, it presents a Graphics Interpreter Language which makes it considerably easier for a user to design his own system, or to add modifications to the current one. This will then enable the user to tailor the process of model description to provide facilities which will be especially useful during model construction.

2. Choice of Data Structure

Since a model description system essentially creates files of models from user input, one of the first things to be designed was the data structure. Due to the small amount of memory, one of the major points to be considered in the following discussion is that the data structure must be simple. If the data structure is fairly involved, too much space will be required to store and manipulate the structure, and the amount which can be used for the data itself will be reduced.

There are essentially three types of information that must be stored:

1) Lines: A line consists of two x-y coordinate pairs, (x_1,y_1) and (x_2,y_2). These two pairs are called an X-Y Quad. Obviously, it takes four words to store an X-Y Quad.

2) Test: Input from the display console keyboard must be stored in the form of text strings. This requires one word of storage for the character count and $\left\lceil \frac{n}{2} \right\rceil$ words for an n character string. (Characters are stored two per word.) Because text strings are usually displayed, two additional words are needed to hold the x-y coordinate pair which positions the line on the screen.

3) Information Words: With each group of data, some words may be required to describe the relationship of this data to the rest of the model description. For example, if a line connects two items, it is usually necessary to store the names of these items with the line coordinates.

The smallest data unit is the entry. It must be able to contain the three types of data described above. However, it is not always necessary to store all three types in every entry. For instance, only

an X-Y Quad is needed to store a line, but, to store the parameters of an element, one might need a text string to list the parameters, and an information word to point to the element they describe. Consequently, some provision must be made to allow an entry to store any combination of the three data types.

One problem, though, with permitting all entries to store any combination of data types is that at least one extra word would be needed in each entry to describe the combination being used. From previous experience, it was found that a picture tends to be stored in groups. A possible picture may consist of a group containing all the lines in the picture, one containing all the text, another containing all the connections, etc. As a result, it seemed that a wise limitation would be to group entries into blocks, within which, all entries must have the same combination of data types. This approach conserves storage space because, now, only a few words are needed in the block header to describe the data combinations, and, in addition, picture storage will tend to be more structured.

Although information words and X-Y Quads may be fixed in length, different text strings have different lengths. For this reason, a word is needed in each entry to specify the total length of the entry. It is, of course, possible to eliminate this word for entries which don't contain text strings, but this makes handling entries more complex. In addition, it was found that, in many cases, the data structure is much more workable if text strings are optional for entries within the same block. (This extra word permits text strings to be optional

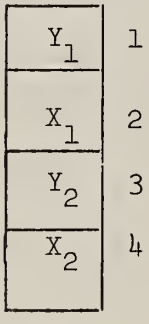
in all entries, because if the length of the entry and length and number of the other two data types is known, the existence and length of a text string can be determined.)

So, it was decided that the block header would contain two words to specify the number of information words and sets of X-Y Quads, and that all entries may or may not contain text strings.

Since the tedious job of displaying a picture or part of a picture is encountered very often, it was decided to make this a "built-in" function, and simply provide display commands. However, in order for this to be done, it is essential that conventions be observed in the storage of data types so that the instructions know where to find the items for display. Consequently, the following storage layout for entries was devised:

1) If there are any Information Words (specified in the block header) they appear directly after the entry length. Since this data type is not displayed, anything in any form may be stored in these words.

2) The Information Words are followed by a specified number (in the block header) of X-Y Quads. Each quad must be stored as follows:



That is, the y coordinates are in the first and third words, and the x coordinates are in the second and fourth.

3) Finally, if there are any text lines, they are stored at the end of the entry in the method described below.

It was found in earlier model description systems that text lines tend to be displayed in lists with one line displayed directly under the previous line. (In this case, the x coordinate remains the same for all lines, and the y coordinate is decreased by a character height as one progresses down the list.) Thus, in an effort to reduce the overhead of associating a coordinate pair with every line, it was decided that if an entry contains more than one line, the lines are assumed to be in a list. Since the coordinates of any line in the list can be calculated from the coordinates of the first line, only two words per entry are needed for x-y coordinates. (If more than one isolated line is to be stored, they must be stored one to an entry. An obvious alternative to the method described is to allow more than one isolated line per entry. This, however, requires 2 coordinate words for every line, and, then, when a list of lines is stored, two words would be unnecessarily wasted to store the coordinates of every line in the list. It is also possible to allot a word per entry to describe whether the lines in the entry are isolated or in a list, but the overhead is then comparable to the original method, and additional work is involved in setting and examining this word.)

Text lines, therefore, are stored in an entry as follows:

Starting with the y coordinate, the coordinates of the first line are stored in the first two words. The text lines are then stored, as a list, in the order in which they are to be displayed. The line or lines in an entry is called a Text Group, and there is only one Text Group per entry.

Because blocks consist of entries with basically the same data type combination, they tend to contain only one aspect of the picture. For example, a line block, text block, parameter block, and connection block might be components of a picture. Since it would be rather cumbersome to have to reference a picture by listing all its blocks, blocks are grouped together into files. As a result, a picture may be referenced or stored by listing one or two files. Finally, any number of blocks may be contained in a file, providing the maximum number of words allocated to a file is not exceeded. (At present, it is 1024_{10})

A detailed description of the Data Structure is provided in section 4.4.

3. Choice of Language and Language Structure

3.1 Interpreter Language

The motivation for this language came about from examining the limitations of the initial approach to model description. Orginally, the entire model description system was written completely in PDP/8 Assembly Language (see [3] and [4]). The system turned out to be extremely large and complex, and, therefore, was difficult to understand, modify, or correct. It is inevitable that a general purpose simulation system will be fairly complex and large. However, it is possible to make the system more understandable and flexible by properly structuring the different independent tasks that the system must perform. First of all, it is important to separate simulation system functions from basic operating system functions. That is, a user programming his own system should not have to concern himself with swapping program segments in and out of core, or scheduling I/O operations. For example, if a user receives a file from an I/O device, he simply wants to be notified that he has received a file, and be provided with its location. He is not concerned about the scheduling problems, the device interrupts, or the various other considerations needed to receive a file. Secondly, it would be very helpful if standard functions associated with a simulation system were provided for the user. This would remove many of the menial programming details, and would greatly increase program readability. The process of displaying a file, for example, is a common but rather lengthy operation. The user, however, could simply be provided with a Display File command which would internally invoke the proper section of code to perform the display.

Based on the above considerations, it was decided that the user would be provided with an interpreter language containing a set of commands which would be complete enough to allow the construction of a sophisticated model description system, yet sufficiently general to avoid unnecessary details. The program constructed from this set of commands would then be executed by a permanent resident system, programmed in PDP/8 assembly language. This system would contain the following:

1) Complete facilities to handle all functions not directly related to the user's program, such as, scheduling I/O and other user programs, core management, and interrupt processing.

2) All of the software necessary to interpretively execute the user's program. This is done by associating each command in the interpreter language with a set of PDP/8 instructions which will perform the function specified by the command. This, then, enables the resident system to execute the user program by examining each command and then performing the associated group of PDP/8 instructions.

This restructuring of the model description system provides a number of advantages. First of all, since each instruction in the user program symbolizes a function which, originally, might have required many PDP/8 assembly instructions, the user programs will be shorter and easier to understand and modify. Errors are also much easier to handle. In the initial construction of the resident system, each section of the system, such as, command decoder, interrupt handler, or any instruction group associated with a command, can be tested separately. Next, when the user is testing his program, each command is interpreted by the resident

system, and, thus, many errors will be detected by the resident system. Another advantage is that, since the commands are implemented in software, the interpreter language may easily be expanded to contain new commands. A new command may be installed simply by assigning it an op-code, and writing the appropriate assembly language routine to perform the desired function. It should also be mentioned that user programs are basically transportable. That is, with a few minor modifications, a user program could run on a different hardware configuration which provided a resident system. Another obvious consequence of transportability is that if the resident system at a particular installation required modification, user programs would be completely unaffected. Finally, on small machines, one major advantage to interpreting commands, as opposed to actually placing the assembly language instructions in line, is that a great deal of space is saved. For example, assume that a Display command can be implemented using n words of assembly language instructions, and that a particular user program uses the Display command m times. Then, if the user program is interpreted, a total of $m + n$ words of core are used. (Assuming that a Display command may be represented by one word in the interpreter language.) However, if the actual instructions are placed in line, a total of $m \cdot n$ words of core are used, and, for $m,n \geq 2$, $m \cdot n > m + n$.

3.2 Types of Variables

The overriding consideration in the design of the language was that it was to be implemented on a small machine, specifically, a PDP/8 which only contains 16K, 12 bit words of main memory, and 262K words of disk

space. Since this small size magnified the usual problems of space and execution time, special care was taken to design the language so that all of the memory space, especially the main memory, will be used efficiently, and that the number of disk references will be reduced to a minimum.

The first decision was to remove all dynamic variables from the user program's instruction stream and to forbid modification of any word in the instruction stream. That is, the program written in the interpreter language must be re-entrant, and all variables which may change their value are stored in a dynamic memory area outside of the program. This provides significant savings of both space and time for the following reasons:

1) A number of console users may be using one model description system. If variables are kept within the model description system, separate copies of the whole system will be needed for each console. However, if all dynamic variables are kept outside the system, all consoles may use the same general system which, then, references the different dynamic variables for each console.

2) If a particular model description system cannot fit entirely in main memory, part of it must reside on the disk, and, therefore, there will be times that a section on the disk must replace a section in main memory. If the program is re-entrant, a great deal of time is saved because there is no need to re-copy the replaced section back out to disk.

3) Finally, as a by-product of separation into dynamic and static memory, error location is improved, because all of the variables which might change are stored together, and may be more easily examined.

Thus, there are two types of variables: static and dynamic. The next two sections discuss the allocation and addressing methods decided upon for each of these variables.

3.2.1 Dynamic Variables

After the dynamic variables had been removed from the instruction stream, the next problem was to determine an efficient method of assigning storage to these variables which used as little main memory as possible, and enabled the variables to be referenced with as few bits as possible. It was decided to allocate the dynamic variables in a stack, because a stack makes it possible to allocate space only for those variables which are currently being used. For example, if it is necessary to save some temporary calculations, they are simply pushed on the stack, and then popped off when needed. These stack locations may then be reused for other variables. Thus, a few stack locations may be used to hold many different temporary variables during the program's execution, and when these locations are not being used, they are not allocated. The stack also works similarly for allocation of local variables for a procedure. If a main routine calls a subroutine, storage on the stack for the local variables of the subroutine is allocated at that time. Then, when the subroutine returns to the main routine, the local variables are popped off the stack. Thus, a number of different subroutines can use

the same locations, if they are not active at the same time. In addition to saving space in dynamic memory, the stack requires very few bits to address its contents, and, therefore, the length of the instructions in static memory is reduced. Since the address of the top of the stack is always known, any level of the stack can be referred to by its level number rather than its absolute address. So, for example, the most recent 32 variables pushed on the stack can be referenced in 5 bits. One final point worth noting is that, in general, the most recently used variables are the variables most likely to be referenced again. So if the variables are kept in a stack the most recently used variables will be at the top of the stack, and there will be a number of times when only a few bits are needed for addressing.

In order to keep track of the location and current size of the stack, there are two pointers associated with the stack. The BP, the bottom (or base) of the stack pointer, holds the starting address of the stack, and the TP, the top of the stack pointer, holds the address of the stack location containing the most currently obtained value. (The address contained in the TP increases as the number of entries in the stack increases. So, if the stack is not empty, $C(TP) \geq C(BP)$ and $C(TP) - C(BP) + 1$ is the current number of variables in the stack, where C(A) means the contents of A.) The operations of placing or removing an item from the top level of the stack are probably the most common stack operations. The terms "push" and "pop" are used to describe these actions and are defined as follows:

Push - A specified item is placed on the top of the stack, and $TP \leftarrow TP + 1$

Pop - The top level of the stack is discarded and

TP ← TP - 1.

In Figure 2, an example has been constructed which shows the status of the stack and its pointers after these operations. Note that the BP always remains constant, and that the TP changes as the stack expands and contracts.

So, in order to determine the actual location of a stack variable, it is only necessary to know which pointer is used as the base address, and the relative position of the variable in the stack with respect to that pointer. (For example, if the TP = 1007 and a stack variable, V, is 4 levels from the top then, assuming stack levels are numbered starting with 0, the address of V is 1007 - 4 + 1 = 1004.) Since either the TP or BP may be used as a base, there are two different kinds of variables:

1) Local variables - are referenced in the stack using the TP as a base address. These variables are associated with a particular procedure (subroutine), and may only be directly referenced by instructions within that procedure. Local variables are allocated only when the procedure is invoked, and are de-allocated when the procedure returns to the calling program. (Note: Through the use of indirect addressing, to be explained later, local variables from one procedure can be passed as parameters to another procedure.)

2) Global variables - are referenced in the stack using the BP as a base. These variables remain allocated throughout the existence of the program, and may be referenced by any instruction in any procedure.

Assume that, initially, there are no entries in the stack, and that the stack starts at core address 1000_8.

Figure 2

The TP is used as a base address for local variables because local variables will naturally tend to be at the top of the stack. Consequently, a savings is gained in the number of bits needed to address a stack variable. However, since the location of the top level of the stack is continually changing, the relative position of a variable from the top will also be changing. Thus, the TP can't be used as a base address for global variables, because a procedure can be called from a number of different places with different stack levels, and if a variable has already been allocated upon entry to a procedure (that is, the variable is global to the procedure), there is no way of knowing, at assembly time, what relative position a variable is from the top of the stack. The bottom of the stack, though, remains constant throughout a program, and, therefore, the relative position of a variable from the bottom of the stack remains constant. So, by using the BP as a base address, global variables can be referenced.

Finally, after it was decided that all dynamic variables would be kept in a stack, the remaining decision was to determine how the address of a referenced variable would be stored in an instruction word. However, before this decision could be made, some estimation of the number of different operations had to be made, because this would determine how many bits would be needed for specifying all operations and would, therefore, also determine the number of bits remaining in the instruction word for addressing. Due to the small number of bits in the PDP/8 word, and the small amount of main memory, it was decided that the instructions referencing dynamic variables would be limited to one specified operand, and that, whenever possible, the opcode and the address of the operand

would be stored in a single word. So, based on these considerations, a skeleton instruction set was drawn up. It was then refined and expanded by writing a number of sample model description systems. From the information gathered, it was decided that a minimum of 5 of the 12 bits in an instruction word would be needed for opcodes. This left 7 bits to specify the address of a dynamic variable. In addition to providing the address length, the sample systems also furnished an estimation of the number of globals and locals used in an average program. Using the above information as a basis, it was decided that, with the 7 bits provided, 64 globals could be referenced directly, and 32 locals could be referenced either directly or indirectly. An extra word would be needed to reference any variables whose level number is above the prescribed limits.

The address calculation method drawn up based on these specifications is presented in the next section.

3.2.1.1 Address Calculation of Dynamic Variables

There are two formats for specifying the addresses of dynamic variables. (See Figure 3.) Format 1 allows an address and an opcode to be specified in one word, while Format 2 provides for addresses which cannot be described in 7 bits. Thus, the basic difference between the two formats is their range of reference. In Format 2, the N part of the address, which specifies the level number, is twice as large as that in Format 1. In both formats, though, the three parts of the address, T,R, and N, function in the same manner.

The type bit, T, is used to specify whether the variable is global or local.

If T = 1, the variable is global, and the entire address (7 or 12 bits) is treated as a negative twos complement number, which designates the number of levels the operand is from the bottom of the stack. The bottom level is -1, and the levels get increasingly negative as the top of the stack is approached. (There is one special case. When Format 1 is being used, if T = 1 and the remaining bits in the address are all zero, then the word immediately following the one containing the current instruction contains the address of the operand in Format 2. This is used to enable instructions to reference stack variables which can't be addressed in 7 bits.)

If T = 0, the variable is local, and the reference bit, R, is then examined to determine if the address is direct or indirect.

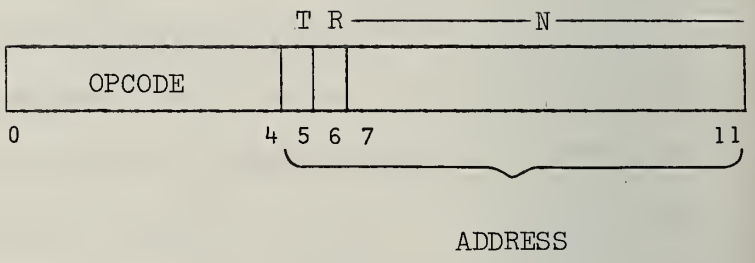
If R = 0, the reference is direct, and the remaining 5 or 10 bits are treated as a positive integer, I. The location of the operand is then found as follows:

If the address were obtained from the instruction stream (direct address), the operand is I levels from the top of the stack. (Levels are numbered starting with 0)

If the address were obtained from the stack (indirect address), the operand is I levels from the level which contained the address.

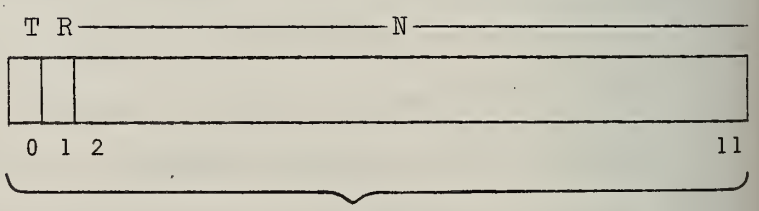
If R = 1, the reference is an indirect reference, and the remaining 5 or 10 bits are regarded as a positive integer, I, which is used

FORMAT 1:



ADDRESS FORMAT IN INSTRUCTION WORD

FORMAT 2:



EXTENDED ADDRESS FORMAT

FIGURE 3

to reference a stack level containing the address of the operand. The position of this stack level is obtained as follows:

If the integer, I, were obtained from an address in the instruction stream, then the new address of the operand is at the Ith level from the top of the stack.

If the integer, I, were obtained from an address in the stack, then the new address of the operand is at the Ith level below the previous address in the stack. (This situation occurs if there is more than one level of indirect addressing.)

In indirect addressing, once the new address is obtained, the whole procedure is then repeated with the new address until a direct reference is made. Notice that any number of indirect references are permitted.

An algorithm for performing the address calculation is presented below.

Description of the variables used in the address calculation algorithm:

TP - contains the address of the current top level of the stack

BP - contains the address of the base (bottom level) of the stack

PC - contains the address of the current instruction to be interpreted

A - will contain the address of the desired operand

X,Y - are temporary variables

The remaining symbols are used to refer to parts of a word:

$T_i(x)$ - If i=1, Format 1 is being used, and $T_i(x)$ refers to bit 5 of the word whose address is contained in x.
If i=2, Format 2 is being used and $T_i(x)$ refers to bit 0 of the word whose address is contained in x.

$R_i(x)$ - If i=1, Format 1 is being used, and $R_i(x)$ refers to bit 6 of the word whose address is contained in x.
If i=2, Format 2 is being used, and $R_i(x)$ refers to bit 1 of the word whose address is contained in x

$N_i(x)$ - If i=1, Format 1 is being used, and $N_i(x)$ refers to bits 7 - 11 of the word whose address is contained in x. If i=2, Format 2 is being used, and $N_i(x)$ refers to bits 2 - 11 of the word whose address is contained in x. When used in an arithmetic calculation with a 12 bit number, $N_i(x)$ is regarded as a twos complement number with the sign bit duplicated to the left the appropriate number of positions.

$TRN_i(x)$ - functions the same as $N_i(x)$, except that if i=1, it refers to bits 5 - 11, and if i=2, it refers to bits 0 - 11.

ALGORITHM:

I (Initialization) X = PC; Y = TP + 1; i = 1

II (See if Global or Local) If $T_i(x) = 0$, go to IV

(Check for special case)

If i = 2, go to III

If ($R_i(x) = 0$ and $N_i(x) = 0$) is not true, go to III

PC = PC + 1

X = PC ; i = 2

Go to II

III ($T_i(x)$ = 1, Global variable)

A = BP - $TRN_i(x)$ - 1

End

IV (See if Direct or Indirect Reference)

If $R_i(x)$ = 1, go to V

(Reference is direct)

A = Y - $N_i(x)$ - 1

End

V (Reference is indirect)

If X = Y, go to VI

X = Y - $N_i(x)$ - 1 ; i = 2

Go to VII

VI X = X - $N_i(x)$ - 1

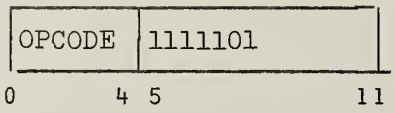
VII Y = X

Go to II (Repeat procedure)

Some examples of address calculation, using the sample stack in Figure 4, are presented next.

Note that BP = 700, and TP = 1017, and assume that each sample instruction is at 2000, i.e., PC = 2000.

Example 1:



Format 1

$T_i(x) = T_1(2000) = 1$. Variable is global.

$TRN_i(x) = TRN_1(2000) = 1111101$ (-3)

$A = BP - TRN_1(2000) - 1$

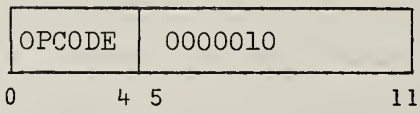
$= 700 - (-3) - 1$

$= 702$

The address of the desired operand in the stack is 702

Thus, consulting Figure 4, the operand is 000010101000 (250_8)

Example 2:



Format 1

$T_i(x) = T_1(2000) = 0$. Variable is local.

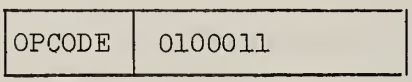
$R_i(x) = R_1(2000) = 0$. Reference is direct.

$N_i(x) = 00010$ (2)

$A = Y - N_i(x) - 1 = TP + 1 - N_1(x) - 1 = 1020 - 2 - 1 = 1015$

The address of the desired operand in the stack is 1015. Consulting Figure 4, the operand is 011001010110 (3126_8)

Example 3:



$T_i(x) = T_1(2000) = 0$. Variable is local

$R_i(x) = R_1(2000) = 1$. Reference is indirect

$X = Y - N_i(x) - 1$ $\quad (N_1(x) = 00011$ (3))

$X = 1020 - 3 - 1 = 1014$

Set i = 2, i.e., use Format 2.

$Y = 1014$

SAMPLE STACK

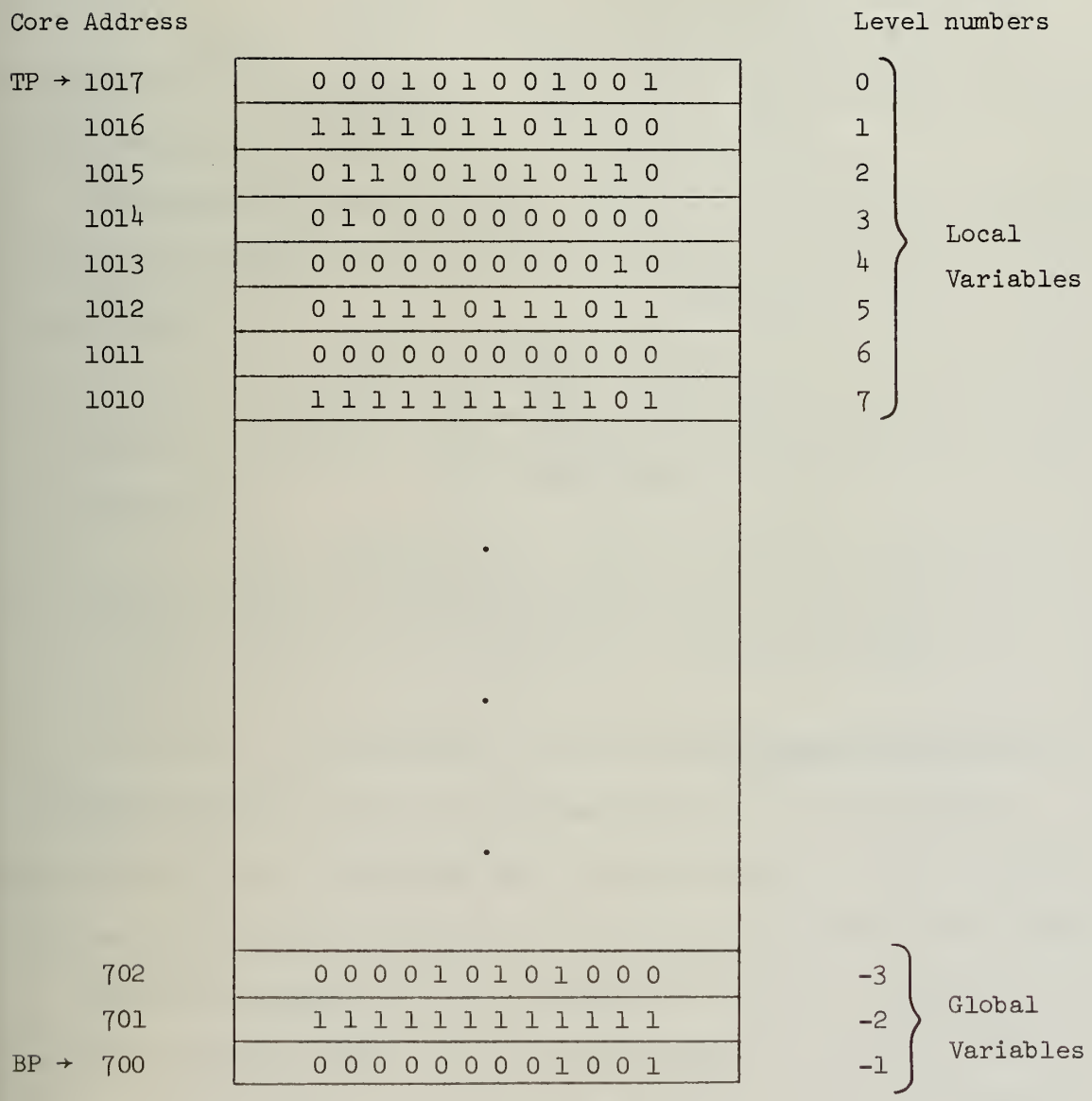


Figure 4 Sample Stack

Repeat Procedure

$T_i(x) = T_2(1014) = 0$. Variable is local

$R_i(x) = R_2(1014) = 1$. Reference is again indirect.

$X = X - N_i(x) - 1$ $\quad (N_2(x) = 0000000000(0))$

$X = 1014 - 0 - 1 = 1013$

$Y = 1013$

Repeat Procedure

$T_i(x) = T_2(1013) = 0$. Variable is local.

$R_i(x) = R_2(1013) = 0$. Reference is direct.

$N_i(x) = N_2(x) = 000000000010$ (2)

$A = Y - N_i(x) - 1 = 1013 - 2 - 1 = 1010$

The address of the desired operand in the stack is 1010.

Consulting Figure 4, the operand is 111111111101 (-3)

3.2.2 Static Variables.

As mentioned above, static variables are variables which remain constant during program execution, and, therefore, can be stored in the instruction stream. This requires that space for these variables must be permanently allocated at assembly time. It seemed that the most effective method of allocation was to store a static variable with the instruction that required it, mainly because no address bits would be needed in the instruction to reference the variable. The main drawback of storing constants with the instruction is that if the same constant is used more than once during the program, it must be stored with each instruction that requires it. The alternative to this would be to store its address in each instruction that uses it. However, this requires that every time any constant is needed, some address bits must be used

to reference the constant. After some study, it was found that in this particular language the number of bits wasted in duplicating constants is far less than the number of bits wasted addressing constants that are only used once. Therefore, it was decided that all constants would be stored with the instructions that reference them.

Since some instructions require constants that don't need a full 12 bit word, two different ways of storing constants with the instruction were chosen:

1) In the instruction - The first word of the instruction contains the constant in the low order 5 to 7 bits, depending on the instruction.

2) After the instruction - If a constant is n words long, it is stored in the n words immediately following any instruction which requires it.

In section 4, a description of the different types of constants will be provided.

4. Instruction Set

The instructions are basically one and two word instructions. (Occasionally, extra words are needed to store a text string or address table with an instruction.) Most instructions do not require an operand, and, in this case, the instruction is a single word with all 12 bits used to represent the opcode. However, if one or more operands are required, the instruction may be greater than one word, and will be composed of an opcode and some combination of the following types of operands:

1) Dynamic Variable Address, DVA: the stack address of a dynamic variable, which is to be used in the execution of the current command, is stored in bits 5 - 11 of the first instruction words. (Bits 0 - 4 contain the opcode.) Dynamic variables and their addressing were discussed in section 3.2.1.1.

For example, if A were the symbolic address of the fourth level of the stack, then ADD A (See description of ADD instruction) would assemble as 1604, and the contents of level 4 would be used in the addition.

2) Literal, L: this is an immediate operand. That is, the operand itself, rather than its address, is contained in the instruction word. A literal is either 5 or 7 bits long, and is always stored in the low order 5 or 7 bits of the first instruction word. (The rest of the first instruction word is used to describe the opcode.)

As an example, consider CSHFT 6. (See description of CSHFT instruction.) This is assembled as 7746, and when the instruction is executed the operand (6) is gotten directly from the instruction.

3) Program Address, PA: A program address is a label associated with a particular instruction.

For storage reasons, an interpreter program is divided into 32_{10} pages of 128_{10} words. (It is possible to have more than one group of 32 pages per program. This is done by using the CALLE instruction, which is described in section 4.6. Note that an instruction may only reference other instructions in its own group, and that one may change groups by using the CALLE instruction.) Any instruction in a group may be referenced by specifying its page, and its position in the page. So, a program address requires one word and is comprised of two parts:

1) Bits 0 - 4: this is a five bit positive integer between 0 and 37_8, specifying the page which contains the word to be referenced.

2) Bits 5 - 11: this is a seven bit positive integer between 0 and 177_8, specifying the location on the page of the referenced word.

For example, if ADDR is the symbolic address for the instruction on page 7, word 123_8, then BR ADDR (see description of BR instruction) would be assembled as:

7601 - Opcode for branch

1723 - Program address

4) Text String, T: A text string is a string of characters of length ≤ 72. If an n character string is to be stored with an instruction, the first word contains the total number of characters in the string, and the remaining $\left\lceil \frac{n}{2} \right\rceil$ words contain the characters stored two to a word in six-bit ASCII. (If there is an odd number

of characters, a blank is stored in the lower half of the last word.) The text string is always the last of the operands to be stored in the instruction.

The instruction, TBAPPT 'ABCDE' (see description of TBAPPT), would be assembled as:

2204	Opcode for TBAPPT
0005	Character count
0102	AB
0304	CD
0540	E

The next sections contain the instructions grouped into major categories: Arithmetic and Logical Commands, Control Commands, Text Buffer Commands, Data Structure Commands, and Input/Output Commands.

Each instruction is presented with the following information:

1) The name of the instruction

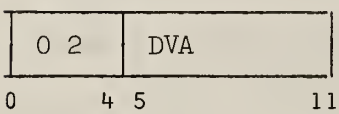
2) The mnemonic of the instruction followed by its operands in their expected order.

3) A layout of the instruction containing the position and octal value of the opcode, and the positions of the operands.

4.1 Arithmetic and Logical Commands

Load Dynamic Variable

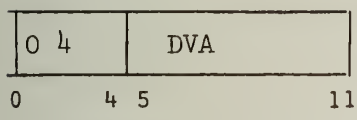
LOAD DVA



A new level is created on the top of the stack, i.e., TP ← TP + 1, and the contents of the stack variable referenced in bits 5 - 11 of the instruction word is loaded into this new level. The contents of the level specified by the DVA remain unchanged. After this instruction has been executed, the total number of levels in the stack has been increased by one. (Note: LOAD 0, assembled as 0200, duplicates the top level of the stack.)

Load Address of a Dynamic Variable

LOADA DVA



A new level is created on the top of the stack, i.e., TP ← TP + 1, and the contents of the instruction word in bits 5 - 11 is loaded into the top level of the stack in the following manner:

Let the bit positions 5 - 11 in the instruction word be represented by $N_5N_6N_7N_8N_9N_{10}N_{11}$.

a) If $N_5 = 0$, then $0N_600000N_7N_8N_9N_{10}N_{11}$ is loaded into the new top level of the stack.

b) If $N_5 = 1$, and $N_6, N_7, N_8, N_9, N_{10}, N_{11} = 0$, then the entire word immediately following the instruction word is loaded, without modification, into the new top level of the stack.

c) Otherwise, if $N_5 = 1$, then $11111N_6N_7N_8N_9N_{10}N_{11}$ is loaded into the new top level of the stack.

This instruction enables addresses stored in the instruction word in Format 1 (or 2) to be loaded onto the stack in Format 2. (Address formats were discussed in section 3.2.1.1.) Note that the first 32 local, and 63 global variables may be loaded onto the stack using only one instruction word. Any variables outside these limits must use 2 words. The first word contains 1000000 in bits 5 - 11, and the second word contains the address of the variable stored in Format 2. One major use of this instruction is to load parameter addresses onto the stack during a subroutine call. (See section 4.6) This instruction is also commonly used to load numbers onto the stack. Obviously, the numbers 0 to 31, and -1 to -63 can be loaded using only one instruction word.

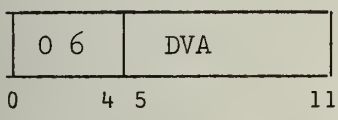
Some examples follow:

1) LOADA LOC where LOC is the name of a local variables which is 3 levels from the top of the stack is assembled as 0402_8. (Since level numbering of locals starts at 0, its level number is 2.) When the instruction is executed 0002_8 is loaded onto the stack. (If the address were to be loaded as an indirect address the instruction would have assembled as 0442_8, and 2002_8 would have been loaded onto the stack.)
2) LOADA GLOB where GLOB is the name of a global variable which is 4 levels up from the bottom of the stack is assembled as 0574_8. When the instruction is executed 7774_8 is loaded onto the stack.
3) LOADA #6 is assembled as 0406_8. When the instruction is executed 0006_8 is loaded onto the stack.

4) LOADA $\#53_{10}$ is assembled as 0500_8 followed by 0065_8. When the instruction is executed 0065_8 is loaded onto the stack.

Store Dynamic Variable

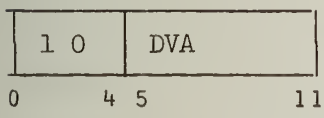
STORE DVA



The top level of the stack is removed (popped) from the stack, i.e., TP ← TP - 1, and stored in the stack level specified by bits 5 - 11 in the instruction word. After this instruction has been executed, the total number of levels in the stack has been decreased by one.

Move Dynamic Variable

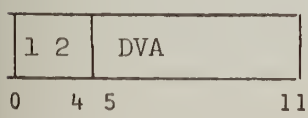
MOVE DVA



The contents of the top level of the stack is stored in the stack level specified by bits 5 - 11 in the instruction word. There is no change in the total number of stack levels. The only difference between the MOVE command and the STORE command is that the STORE command pops the stack.

Inclusive Or

OR DVA



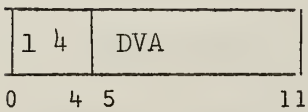
The contents of the stack level referenced by bits 5 - 11 in the instruction word and the contents of the top level of the stack are logically summed. The result is placed in the top level of the stack. The previous contents of the top level are destroyed. There is no change in the total number of stack levels.

If, however, bits 5 - 11 of the instruction word are all zero, the top two levels are popped off the stack, logically summed together, and the result is pushed onto the stack. In this case, the total number of stack levels is decreased by one.

A logical sum (OR) of two operands is formed by comparing the corresponding bit positions of the two operands. If either one or both of the bit positions in the operands contains a one, the corresponding bit position in the result is set to one; otherwise it is set to zero.

And

AND DVA



The contents of the stack level referenced by bits 5 - 11 in the instruction word and the contents of the top level of the stack are combined to form a logical product. The result is placed in the top level of the stack. The previous contents of the top level is destroyed. There is no change in the total number of stack levels.

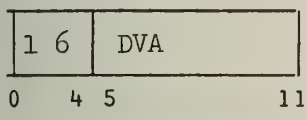
If, however, bits 5 - 11 of the instruction word are all zero, the top two levels are popped off the stack, their logical product is formed,

and then pushed onto the stack. In this case, the total number of stack levels has been decreased by one.

A logical product (AND) of two operands is formed by comparing the corresponding bit positions of the two operands. If both of the bit positions of the two operands contain a one, the corresponding bit position in the result is set to one; otherwise it is set to zero.

Add

ADD DVA

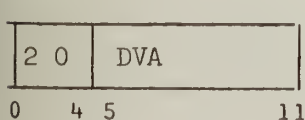


The contents of the stack level referenced by bits 5 - 11 in the instruction word and the contents of the top of the stack are added together using twos - complement addition and the result is placed in the top level of the stack. The previous contents of the top level are destroyed. There is no change in the total number of stack levels.

If, however, bits 5 - 11 of the instruction word are all zero, the top two levels are popped off the stack, added together, and their sum is pushed onto the stack. In this case, the total number of stack levels has been decreased by one.

Compare (Arithmetic)

CMP DVA



The contents of the stack level referenced by bits 5 - 11 in the instruction word and the contents of the top level of the stack are treated as twos - complement numbers, and are arithmetically compared using the rules for twos - complement arithmetic.

a) If C(Top Level) > C(DVA), a + 1 is pushed onto the top of the stack.

b) If C(Top Level) = C(DVA), a 0 is pushed onto the top of the stack.

c) If C(Top Level) < C(DVA), a - 1 is pushed onto the top of the stack.

The contents of both operands remain unchanged. After execution, the total number of stack levels has been increased by one.

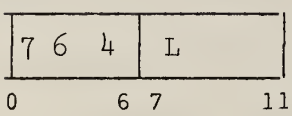
This command may be used in conjunction with the branch instructions described in the next section.

For example,

```
    CMP A     Compare the contents of level A with the top level
BMZP ADDR     If the contents of the top level were less than or
              equal to the contents of level A, branch to ADDR.
              Pop the stack after execution of the instruction.
```

Pop

POP L

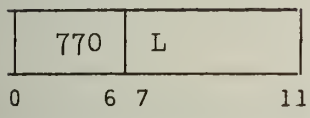


Bits 7 - 11 of the instruction word are treated as a 5 bit positive integer, L. The top L levels are then removed (popped) from the stack, and TP ← TP - L.

If L = 0, then the top level is popped and examined (as a 12 bit positive integer) to determine how many additional levels must be popped. Therefore, if L = 0, TP ← TP - (C(Top Level) + 1).

Arithmetic Shift

ASHFT L



Bits 7 - 11 of the instruction word are treated as a 5 - bit twos - complement integer, L. The top of the stack is then arithmetically shifted L bit places. If L is positive, the shift is to the right; if negative, to the left. In an arithmetic shift to the right, the bits shifted off the right end are ignored, and the high order bit is duplicated on the left. In a shift to the left, the bits shifted off the left are ignored, and the low order positions which were vacated by the original bits are filled with zeros.

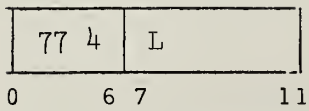
If L = 0, the top of the stack is popped and the contents are treated as a twos- complement integer which determines how many positions the new top level is to be shifted.

If L ≠ 0, there is no change in the total number of levels

If L = 0, TP ← TP - 1

Circular Shift

CSHFT L



Bits 7 - 11 of the instruction word are treated as a 5-bit twos - complement integer, L. The top of the stack is then circularly shifted L bit places. If L is positive, the shift is to the right; if negative, to the left. In a circular shift, each bit that is shifted out of the word on one end is stored in the vacant position that was created on the other end.

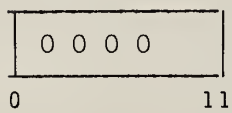
If L = 0, the top of the stack is popped and the contents are treated as a twos - complement integer which determines how many positions the new top level is to be shifted.

If L $\neq$ 0, there is no change in the total number of levels.

If L = 0, TP $\leftarrow$ TP - 1.

No Operation

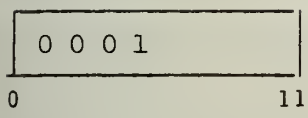
NOOP



No operation is performed. This instruction is primarily used by the assembler to fill up a program page which doesn't have enough words to store the next instruction.

Negate the Top of the Stack

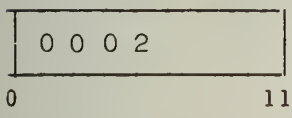
NEG



The contents of the top level of the stack are negated.

Complement the Top of the Stack

COM



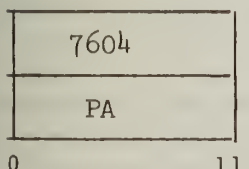
The contents of the top level of the stack are complemented. All bits that were originally zero are set to one, and those that were originally one are set to zero.

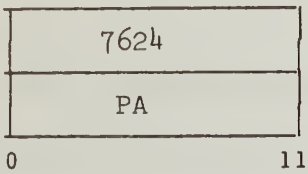
4.2 Control Commands

All branch instructions which end with the letter P denote that, regardless of the success or failure of a conditional branch, the top word on the stack is popped after the instruction is executed.

Branch on Zero

BZ PA (or) BZP PA

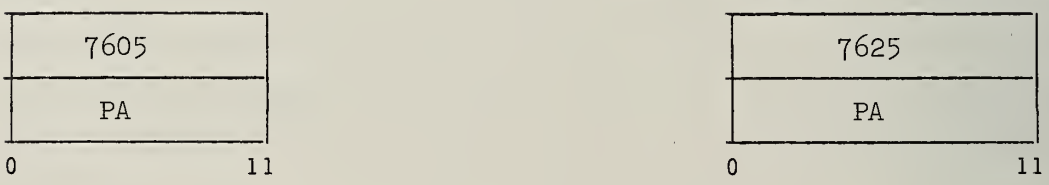




If the top word on the stack is zero, the program branches to the program address specified. The BZP instruction functions exactly as the BZ instruction, except that after execution of the BZP instruction, the top of the stack is popped.

Branch on Non-Zero

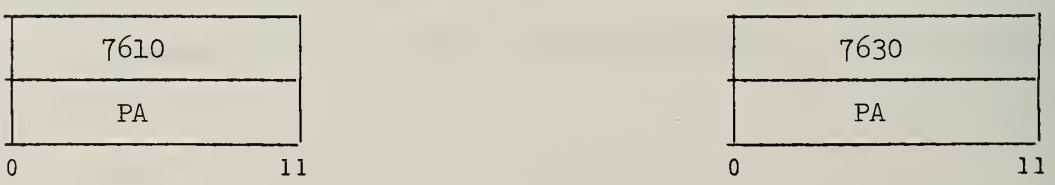
BNZ PA (or) BNZP PA



If the top word on the stack is not zero, the program branches to the program address specified. The BNZP instruction functions exactly as the BNZ instruction, except that after execution of the BNZP instruction, the top of the stack is popped.

Branch on Minus

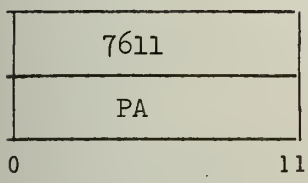
BMI PA (or) BMIP PA

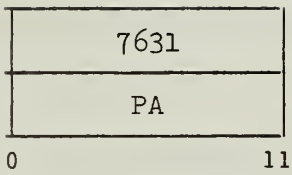


If the top word on the stack is negative, that is, bit 0 is a 1, the program branches to the program address specified. The BMIP instruction functions exactly as the BMI instruction, except that after execution of the BMIP instruction, the top of the stack is popped.

Branch on Plus or Zero

BPZ PA (or) BPZP PA

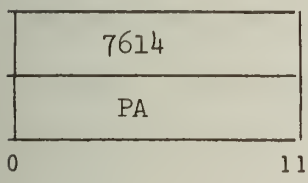


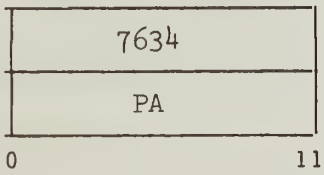


If the top word on the stack is greater than or equal to zero, the program branches to the program address specified. The BPZP instruction functions exactly as the BPZ instruction, except that after execution of the BPZP instruction, the top of the stack is popped.

Branch on Minus or Zero

BMZ PA (or) BMZP PA





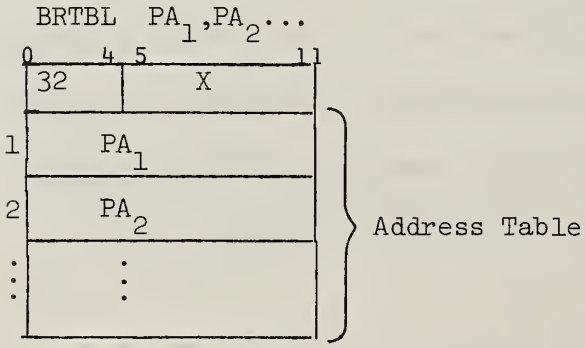
If the top word on the stack is less than or equal to zero, the program branches to the program address specified. The BMZP instruction functions exactly as the BMZ instruction, except that after execution of the BMZP instruction, the top of the stack is popped.

Branch Unconditionally

BR PA (or) BRP PA

The program branches to the program address specified. The BRP instruction functions exactly as the BR instruction, except that after execution of the BRP instruction, the top of the stack is popped.

Branch on Table



The top of the stack is popped, and this number, N, is considered to be a 12 bit twos complement number, and is used as an index into the address table which starts with the second instruction word.

If,

$0 < N \leq$ Number of addresses in the table,

the program branches to the address specified, PA_N, in the Nth instruction word.

If,

$N \leq 0$ or $N >$ Number of addresses in the table,

the next instruction in line is executed.

Bits 5 to 11 of the first instruction word are used to hold a count of the total number of addresses in the table. (This number is set by the assembler.)

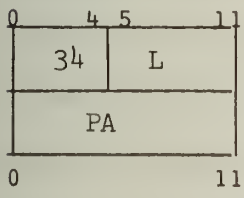
The BRTBL instruction may be used with compare instructions:

CMP WORD	Compare to top level of stack
ADD 2	a - 1, 0, or 1 was placed on the stack, and is now converted to 1, 2 or 3.
BRTBL LT, EQ, GT	The program branches to LT if top level < WORD, to EQ if equal, and to GT if top level > WORD.

The BRTBL instruction may also be used with the GRDNUM instruction (see Section 4.5.2.2) to enable the program to branch to different areas of code, depending on where a joystick hit was made on the display screen. This will be explained more fully later.

Test and Branch

TSTBR L, PA



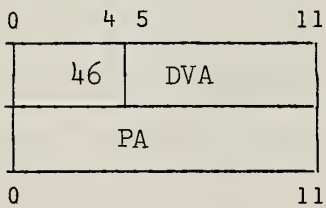
The contents of bits 5 - 11 in the first instruction word is treated as a 7 bit positive integer, L, and is compared with the top level of the stack. (For the comparison, L is assumed to have five high order zeros. The numbers are then compared bitwise.)

If the numbers are equal, the program branches to the address specified in the second instruction word, and the stack is popped. If they are not equal, the next instruction in line is executed, and the stack is not popped.

This instruction is used most often with the WAIT instruction, (See section 4.5.1) to determine whether input was received from a particular device. The devices, such as, IBM 360, PDP10, Display Screen, are numbered. Thus, for example, to quickly determine if input were received from device number seven, it is simply necessary to do a TSTBR #7, ADDR. The instruction will then only branch to ADDR if 7 were the interrupting device number on top of the stack. This will be explained more fully in a later section.

Decrement Variable and Branch if not Zero

DVBNZ DVA, PA



The contents of the stack level referenced by bits 5 - 11 in the first instruction word is decremented by one. If the new contents is zero, the next instruction in line is executed. If the new contents is not zero, the program branches to the address specified in the second instruction word.

This instruction is used primarily for controlling the number of executions of a program loop.

For example, consider the following section of code which adds the numbers 1 to 10:

```
        LOADA #10
        STORE COUNT              Save number of times through loop
```

```
        LOADA #0
LOOP:   ADD  COUNT
        DVBNZ COUNT, LOOP       Decrement count and produce next
              :                 number to be added.
              :
              :
```

4.3 Text Buffer Commands

The Text Buffer, abbreviated TB, is used to process and examine text strings. It consists of a header and a 72 character buffer. These are described in Figure 5.

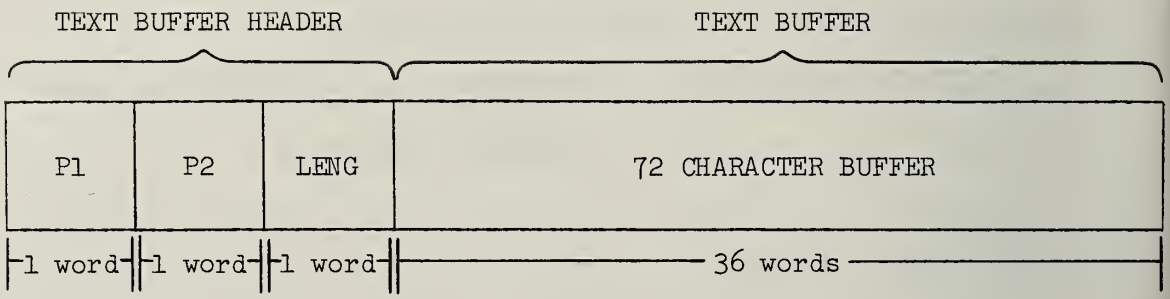
The instructions described below enable the user to operate on the contents of the TB through the use of the pointers P1 and P2. However, certain conventions must be observed regarding these pointers.

When it is desired to refer to a particular text string in the TB, P1 should be positioned to point to the first character in the string, and P2 should be positioned to point to the character position immediately following the last character in the string. Thus, whenever P1 and P2 point to a text string:

a) P1 < P2 (If not an error is given.)

b) The text string is P2 - P1 characters long, and contains the characters starting at the position pointed to by P1, up to and including the position pointed to by P2-1.

Although the pointers P1 and P2 may be moved anywhere in the TB through use of the TBMOVE instructions, it is advisable to keep P1 $\leq$ P2.



P1, P2 are pointers which contain the number of the character they are pointing to in the buffer. The character buffer is numbered from left to right starting with 1 ($1 \leq P1, P2 \leq LENG + 1$). These pointers are set by various text buffer instructions.

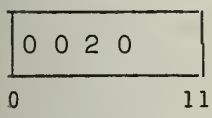
LENG contains the length, in characters, of the text string contained in the buffer.

TEXT BUFFER

FIGURE 5

Initialize Text Buffer

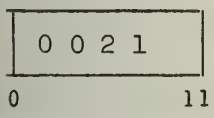
TBINIT



The text buffer is initialized by setting P1, P2 = 1 and LENG = 0.

Push Character onto Stack

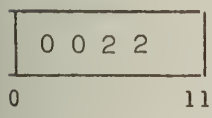
TBPUSHC



The character pointed to by P1 is pushed onto the stack in the six low order bit positions. The six high order bit positions are filled with zeros. (The character is not deleted from the TB.) P1, P2 and LENG are unchanged.

Insert Character into Text Buffer

TBINSC



The character pointed to by P1 and all of the characters to its right are shifted one character position to the right. Next, the six low order bits of the word on the top level of the stack are inserted in the TB at P1. Finally, the top level of the stack is popped off the stack. P1 remains the same, i.e., it points to the inserted character.

LENG' = LENG + 1

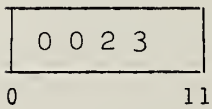
If P1 $\leq$ P2, P2' = P2 + 1

If P1 > P2, P2' = P2

This adjustment of P2 is done to ensure that P2 is pointing to the same character before and after the instruction.

Delete a Character from the Text Buffer

TBDELC



The character pointed to by P1 is deleted. The character at P1 + 1 and all of the characters to its right are shifted to the left one character position to fill the vacancy.

P1 remains the same.

LENG' = LENG - 1

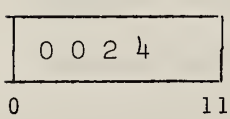
P2 is calculated as follows:

If P1 < P2, P2' = P2 - 1

If P1 $\geq$ P2, P2' = P2

Change a Character in the Text Buffer.

TBCHGC



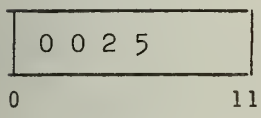
The six low order bits of the word on the top level of the stack are stored in the TB at the character position pointed to by P1, and then the top level of the stack is popped. (Whatever character was previously at

the position pointed to by P1 is destroyed.)

P1, P2 and LENG are unchanged.

Append Character to the Text Buffer

TBAPPC



The six low order bits of the word on the top level of the stack are appended to the end of the text string in the TB at the character position LENG + 1, and then the top of the stack is popped.

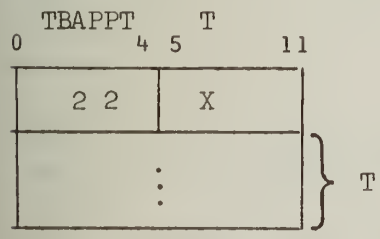
P1 is unchanged.

LENG' = LENG + 1

If P2 = LENG + 1, P2' = P2 + 1, otherwise P2' = P2.

Thus, if P2 is pointing to the end of the buffer before the instruction, it will be moved to point there afterward.

Append Text String onto Text Buffer



The text string, T, is appended to the end of the TB starting at character position LENG + 1.

Bits 5 to 11 of the first instruction word are used to hold a count of the number of words needed in the instruction to represent the text string, T. (This number is determined by the assembler.)

Note that if no text string is specified in the instruction, bits 5 to 11 are set to zero. Then, rather than immediately following the first instruction word, the text string is assumed to be in the current entry at the line number contained in the LN. (See Data Structure Commands.)

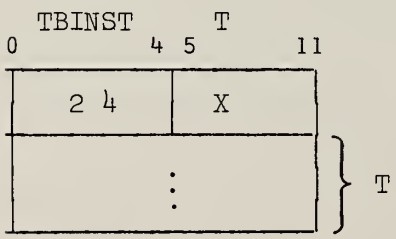
P1 is unchanged.

If the number of characters in the text string is M, then P2 and LENG are calculated as follows:

If P2 = LENG + 1, P2' = P2 + M, otherwise P2 is unchanged.

LENG' = LENG + M

Insert Text String into Text Buffer



Assume M is the number of characters in the text string, T. Then the character pointed to by P1 and all of the characters to its right are shifted to the right M positions. The text string, T, is inserted into the TB at positions P1 to P1 + M - 1.

Bits 5 to 11 of the first instruction word are used to hold a count of the number of words needed in the instruction to represent the text string, T. (This number is determined by the assembler.)

Note that if no text string is specified by the user, bits 5 to 11 are set to zero. Then, rather than immediately following the first instruction word, the text string is assumed to be in the current entry at the line number contained in the LN (See Data Structure Commands).

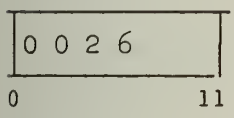
P1 is unchanged

If P1 $\leq$ P2, P2' = P2 + M. Otherwise, P2' = P2.

LENG' = LENG + M

Delete Text String from Buffer.

TBDELT



All characters in the TB between P1 and P2 - 1 are deleted. The character at P2 and all characters to its right are shifted to the left P2 - P1 character positions.

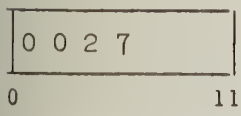
P1 remains the same

P2' = P1

LENG' = LENG - (P2 - P1)

Convert Number to Text String

TBCNV



The word on the top level of the stack is treated as a twos - complement number, and is converted to a base 10, 6 - bit ASCII text string.

(Insignificant leading zeros are removed.) It is then appended to the end of the text string in the TB at character position LENG + 1. Finally, the top level of the stack is popped.

P1 is unchanged.

If the number of characters in the converted text string is M, then P2 and LENG are calculated as follows:

If P2 = LENG + 1, P2' = P2 + M; otherwise, P2 is unchanged

LENG' = LENG + M

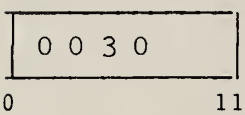
Some examples of conversions:

	Top of the stack		ASCII conversion (base 10)		
a)	000 000 010 001	(0021_8)	61 | 67	(17_{10})	M = 2
b)	111 111 100 101	(-0033_8)	55 | 62 | 67	(-27_{10})	M = 3
c)	000 000 000 000	(0000_8)	60	(0_{10})	M = 1

The converted ASCII string in each of the examples would be appended to the TB at LENG + 1.

Compress the Text String

TBCMPRS



All blank characters in the text string pointed to by P1 and P2 are deleted, and the text string is shifted left the appropriate number of character positions.

P1 remains unchanged.

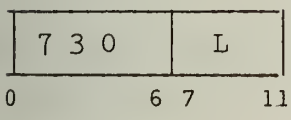
If the total number of blanks in the text string were equal to M, then P2 and LENG are calculated as follows:

P2' = P2 - M

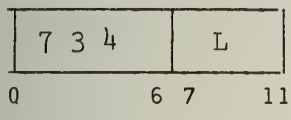
LENG' = LENG - M.

Move Pointer

TBMOVEP1 L



TBMOVEP2 L



The pointer, P1 or P2, depending on which instruction is being executed is moved according to the value of bits 7 to 11 of the instruction word. (P will be used to represent the pointer specified in the instruction.) Bits 7 to 11 are treated as a 5 bit sign-magnitude integer, L. Movement of P is to the right if L is positive, and to the left, if L is negative. The possible values of L and their meanings are described below. If a particular value of L is followed by a symbol in parentheses, that symbol may be used in place of the number when using the instruction. (If movement is to the left, the symbol must be preceded by a -.)

If,

L = +0 the top of the stack is popped and the contents are treated as a 12 bit sign-magnitude number.

This number then specifies how many positions P is to be moved. (This enables P to be moved large distances.)

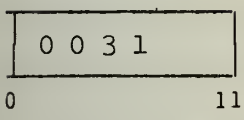
L = -0 ($P)	The pointer, P, is moved to the character position of the other pointer. Thus, if P = P1, P' = P2 and if P = P2, P' = P1.
L = $\pm 1_8$ to 12_8	The pointer, P, is moved |L| positions, i.e., P' = P $\pm$ |L|
L = $\pm 13_8$ ($B)	The pointer, P, is moved to the next blank character
L = $\pm 14_8$ ($NB)	The pointer, P, is moved to the next non-blank character.
L = $\pm 15_8$ ($A)	The pointer, P, is moved to the next alphanumeric character.
L = $\pm 16_8$ ($NA)	The pointer, P, is moved to the next non-alphanumeric character.
L = $+17_8$ ($E)	The pointer, P, is moved to the end of the text string, i.e., P' = LENG + 1.
L = -17_8 ($S)	The pointer, P, is moved to the start of the TB, i.e., P' = 1.

If P is to be moved to a particular type of character, such as, blank, non-blank, alphanumeric, or non-alphanumeric, and a character of that type does not appear in the string, P is not moved at all. So, the next instruction should normally test to see if P is pointing to the type of character desired.

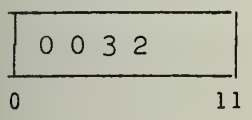
The LENG and other pointer are unaffected by this instruction.

Load Pointer.

TBLDP1



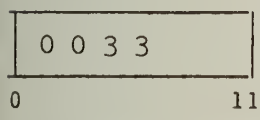
TBLDP2



The contents of the pointer specified by the instruction is pushed onto the stack. (This will be a number between 1 and 73_{10}).

Load the Length of Text String

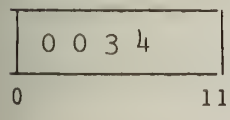
TBLDLENG



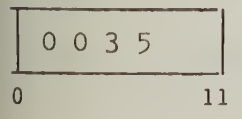
The length of the text string currently in the TB is pushed onto the stack. (This will be a number between 1 and 72_{10}.)

Store Pointer

TBSTP1



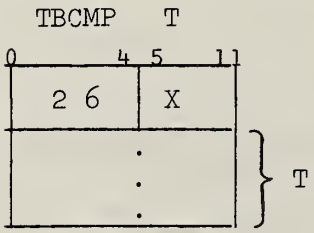
TBSTP2



The contents of the top word of the stack is stored in the pointer specified by the instruction. The top of the stack is then popped.

It is an error if the word, W, to be stored as P1 or P2 does not satisfy: $1 \leq W \leq LENG + 1$

Compare Text String



The text string, T, is compared with the string in the TB pointed to by P1 and P2.

If T > TB a 1 is pushed onto the stack

If T = TB a 0 is pushed onto the stack.

If T < TB a -1 is pushed onto the stack.

The type of comparison used is the following:

The two operands are left justified and then, starting with the left-most character, are compared numerically, character by character, as 6 bit positive integers. (Both strings are represented in 6 bit ASCII.) If the operands are of unequal length, they are compared until the end of the shorter. If they are equal up to this point, the longer of the two is declared to be greater.

Bits 5 to 11 of the first instruction word are used to hold a count of the number of words needed in the instruction to represent the text string, T. (This number is determined by the assembler.)

If no text string is specified by the user, bits 5 to 11 are set to zero. Then, rather than immediately following the first instruction word, the text string is assumed to be in the current entry at the line number contained in the LN. (See Data Structure Commands.)

P1, P2 and LENG remain unchanged.

Some examples,

a) TB : ABC T > TB, a 1 is pushed onto the
   T : ABCD stack.

b) TB : BCD TB > T a 1 is pushed onto the
   T : ABCDE stack.

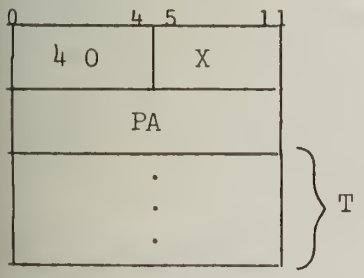
c) TB : ABC T > TB a 1 is pushed onto the
   T : ABC stack.

d) TB : BCD T > TB a 1 is pushed onto the
   T : BC; stack.

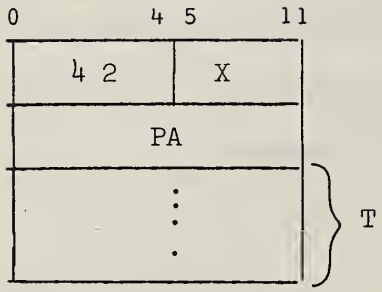
After the TBCMP instruction, the branch instructions may be used as described in the CMP instruction.

Match Text String

TBMATP1 T,PA



TBMATP2 T,PA



The text string, T, is compared with the string of equal length in the TB starting at the pointer specified in the instruction. (P will be used to represent the specified pointer.)

If the text strings are unequal, the next instruction in line is executed, and P1, P2 and LENG remain unchanged.

If the text strings match, P is pointed to the character position immediately to the right of the matched string in the TB, and the program branches to the specified address, PA. The other pointer and LENG remain unchanged.

Bits 5 to 11 of the first instruction word are used to hold a count of the number of words needed in the instruction to represent the text string, T. (This number is determined by the assembler.)

If no text string is specified by the user, bits 5 to 11 are set to zero. Then, rather than immediately following the first instruction word, the text string is assumed to be in the current entry at the line number contained in the LN. (See Data Structure Commands.)

Some examples,

Contents of TB:

P1	P2	LENG	
1	7	6	ABCDE

a) Before: TBMATP1 'BCD', ADDR

After: No match. Next instruction is executed

1	7	6	ABCDE

b) Before: TBMATP1 'ABCD', ADDR

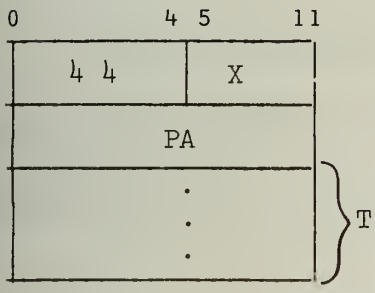
P1	P2	LENG	
1	7	6	ABCDEF

After: Match. Instruction at ADDR is executed.

5	7	6	ABCDEF

Search for Text String

TBSEARCH T,PA



The text string pointed to by P1 and P2 is searched for the first occurrence of the text string, T.

If T is not found, P1, P2 and LENG remain unchanged, and the next instruction in line is executed.

If T is found, P1 and P2 are set to point to it, LENG remains unchanged, and the instruction at the specified address, PA, is executed next.

Bits 5 to 11 of the first instruction word are used to hold a count of the number of words needed in the instruction to represent the text string T. (This number is determined by the assembler.)

If no text string is specified by the user, bits 5 to 11 are set to zero. Then, rather than immediately following the first instruction word, the text string is assumed to be in the current entry at the line number contained in the LN. (See Data Structure Commands)

Some examples,

Contents of TB

a) Before: TBSEARCH 'ABC', ADDR

P1	P2	LENG	
1	5	4	CDEF

After: Next instruction in line is executed

1	5	4	CDEF

b) Before: TBSEARCH 'BC', ADDR

P1	P2	LENG	
4	8	7	ABCDEFG

After: Next instruction in line is executed

4	8	7	ABCDEFG

because 'BC' is not between P1 and P2.

c) Before; TBSEARCH 'BCD', ADDR

P1	P2	LENG	
1	7	6	ABCDEF

After: Instruction at ADDR executed

2	5	6	ABCDEF

d) Before: TBSEARCH 'CDE', ADDR

P1	P2	LENG	
1	5	7	ABCDEFG

After: Next instruction in line is executed,

1	5	7	ABCDEFG

because 'CDE' is not completely within P1 and P2 -1.

4.4 Data Structure Commands

Before proceeding with this section, it would be helpful to reread Section 2.

The following discussion refers to Figures 6, 7 and 8.

There are three basic units in the data structure: files, blocks, and entries. A file, pictured in Figure 6, simply contains a file header followed by any number of blocks. The file header is used almost entirely by internal routines, and only the current block number, BN, may be modified by the user.

A block contains a block header followed by any number of entries. The block header is accessible to the user, and is initialized when the block is created. It describes the type of information contained in each of its entries. A description of the block header follows: (See Figure 7)

Word 0 : contains the pointer to the next block in this file. (May not be modified by the user.)

Word 1 : contains the number assigned to this block.

Word 2 : contains the number of the current entry being referenced. This is called the current entry number, EN. (For reference purposes, entries in a block are assumed to be numbered in order starting with 1.)

Word 3 : contains the number of the current line being referenced in the Text Group of the current entry. This is called the current line number, LN. (Lines in a Text Group are assumed to be numbered in order starting with 1.)

Word 4 : contains the number of Information Words in each entry. (See description of entry below.)

Word 5 : contains the number of X-Y Quads in each entry. (See below.)

Word 6 : contains the offset, DY, which is used in conjunction with the y coordinate of any item to calculate the current display y - coordinate.

Word 7 : contains the offset, DX, which is used in conjunction with the X coordinate of any item to calculate the current display X - coordinate.

The offset words enable the contents of a block to be moved simply by changing two words. For instance, if a picture is centered at (X_0,Y_0), it may be moved to $(X_0 + dx, Y_0 + dy)$ by storing dy and dx in the DY and DX offset words. The DY and DX affect anything that is to be displayed in the

FILE

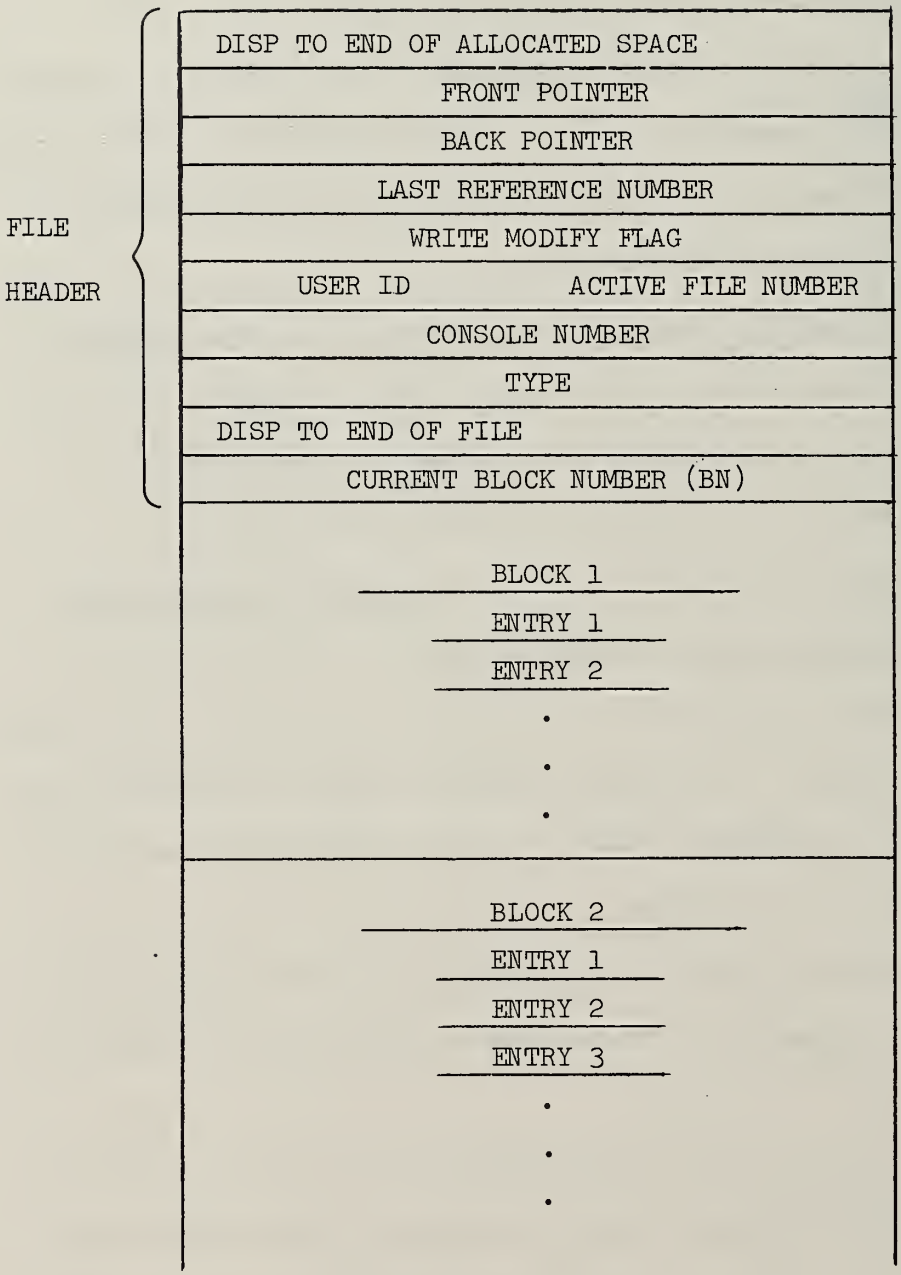
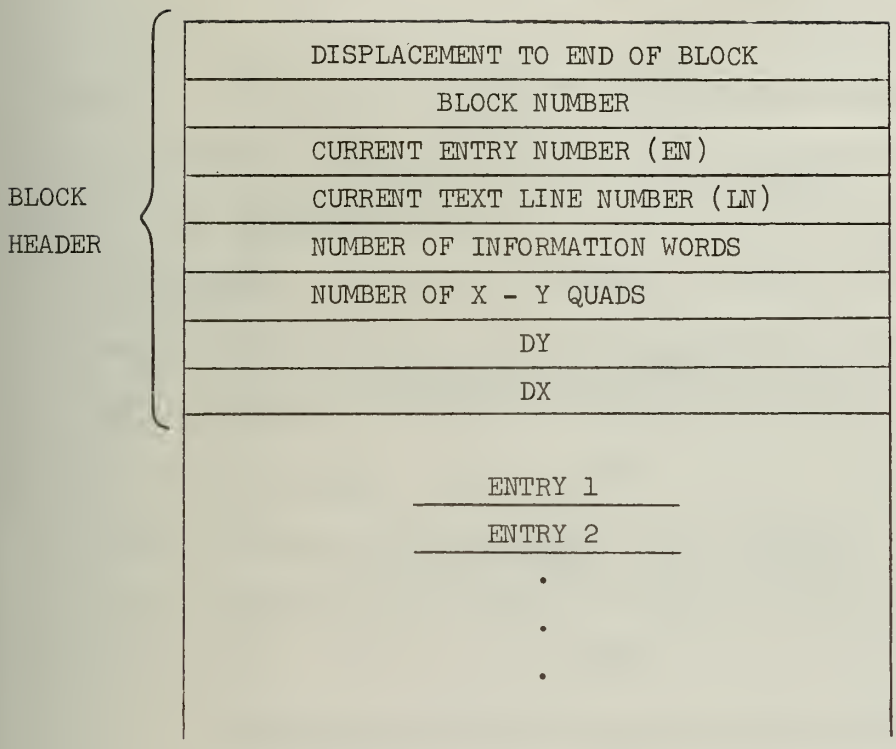


FIGURE 6

BLOCK



ENTRY

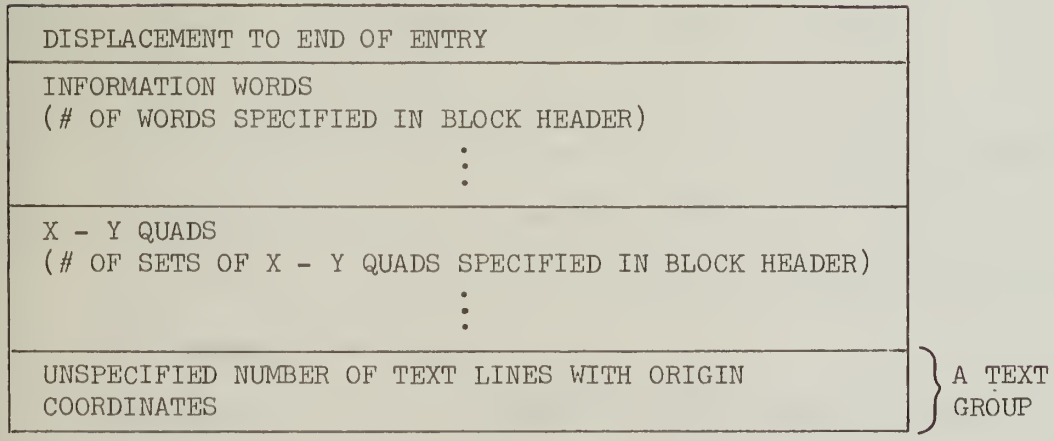


FIGURE 7

TEXT GROUP

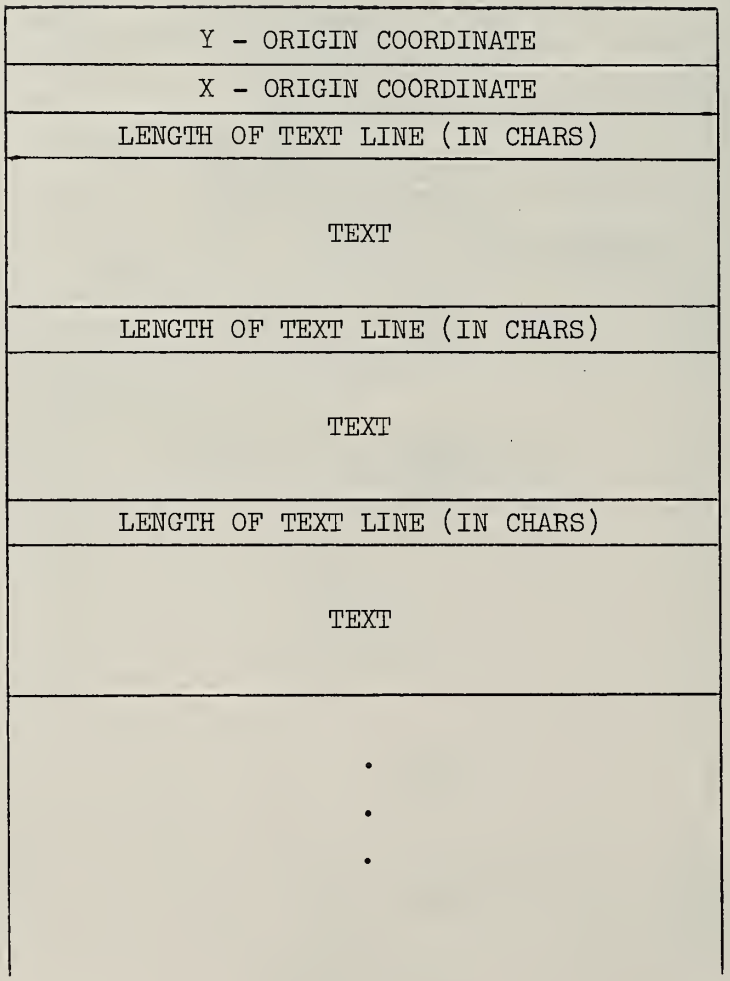


FIGURE 8

entire block. Thus, when displaying, the coordinates of any item in a block are essentially (x + DX, y + DY).

Entries are the structures which contain the bulk of the user's information. They can contain any or all of the following types of data: (See Figures 7 and 8.)

1) Information Words - These are words which are allocated in an entry, and are used to hold any information which can not easily be contained in any of the other types. The number of these words allocated per entry is contained in the block header.

2) X-Y Quads - These are sets of words (four per set) which are used to contain x-y coordinates. Notice that one set corresponds to the number of words needed to describe a single line. The number of these sets per entry is contained in the block header.

3) Text Group:- This is a group of text lines. If the text group contains N lines, they are stored as follows: (See Figure 8)

The first two words of the text group contain the y and x coordinates, respectively, of the leftmost character of the first line in the group. The remainder of the words in the group contain the N text lines with each line being preceded by a word containing its length in characters. The text lines are stored in 6 bit ASCII (two characters per word), and may not exceed 72 characters.

If a Text Group is to be displayed, the first line is displayed at the the specified x-y coordinates (plus offset), and each succeeding line is displayed directly below the one preceding it. Thus, if the first line is displayed at (x + DX, y + DY), the second line will be displayed at (x + DX,

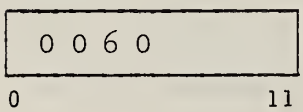
y - hy + DY), where hy is the height of a character, and DX, DY are the offset words. This implies that the coordinates of the n-th line in a group are given by (x_0 + DX, y_0 - (n-1)hy + DY), where y_0, x_0 are the coordinates of the first line in the group.

When an entry is created, space is not preallocated for the Text Group, as it is for the Information Words and X - Y Quads. So, when a line is to be added to the Text Group, its length is determined and space is allocated just before insertion. Thus, for a particular block, the total number of lines in each Text Group may vary from entry to entry, but all entries have the same number of Information Words and sets of X - Y Quads.

4.4.1 File Commands

Set File Number

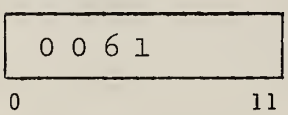
SETFN



The contents of the top level of the stack is stored in the FN. The top of the stack is then popped.

Read File Number

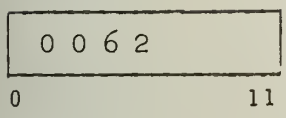
RDFN



A new level is created on the top of the stack, i.e., TP ← TP + 1, and the contents of the FN is stored in this level.

Initialize a File

INITF



An active file is created, and the BN in this file is set to zero. The contents of the top level of the stack is stored in the FN, the newly created file is assigned this number, and the stack is popped.

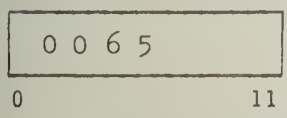
If there were an old file still assigned to the same number, the old file is destroyed.

4.4.2 Block Commands

All block commands refer to a block contained in the file pointed to by the FN.

Set Block Number

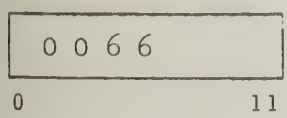
SETBN



The contents of the top level of the stack is stored in the BN. The top of the stack is then popped.

Read Block Number

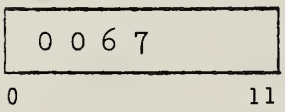
RDBN



A new level is created on the top of the stack, i.e., TP ← TP + 1, and the contents of the BN is stored in this level.

Initialize a Block

INITB



A block, consisting of just a block header, is created and appended onto the current file whose number is in the FN. The BN of the file is set to point to the new block. The EN, LN, DX, and DY in the new block are set to zero. The rest of the block header is initialized according to information which is assumed to be in the top three levels of the stack:

Level 2: contains the number to be assigned to this block.

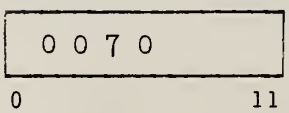
Level 1: contains the number of information words to be assigned to each entry in this block.

Level 0: contains the number of sets of X - Y quads to be assigned to each entry in this block.

After the block header is initialized, the top three levels of the stack are popped, i.e., TP ← TP - 3.

Delete Block

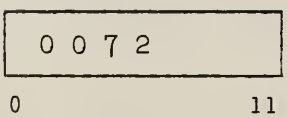
DELB



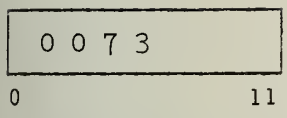
The block whose number is in the BN is deleted. The BN is set to point to the block immediately following the deleted block.

Load Offset Word (DY or DX)

LDDY



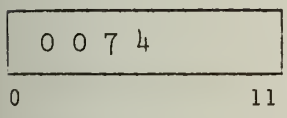
LDDX



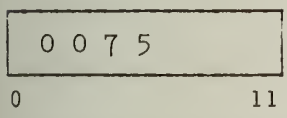
The offset word in the block header of the block whose number is contained in the BN is pushed onto the stack.

Store Offset Word

STDY



STDX



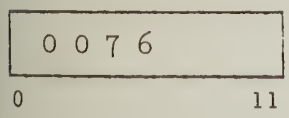
The contents of the top level of the stack is stored in the offset word of the block whose number is contained in the BN.

4.4.3 Entry Commands

All entry commands refer to the entry contained in the block pointed to by the BN, which is contained in the file pointed to by the FN.

Set Entry Number

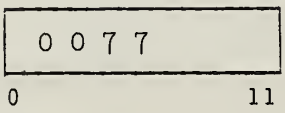
SETEN



The top level of the stack is examined. If the contents is not equal to -1, it is stored in the EN. If it is equal to -1, the number of the last entry in the block is stored in the EN.

Read Entry Number

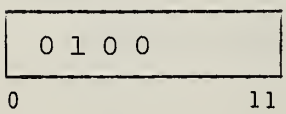
RDEN



A new level is created on the top of the stack, i.e., TP ← TP + 1, and the contents of the EN is stored in this level.

Append an Entry

APPE

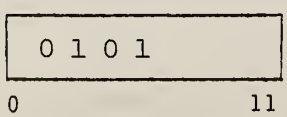


An entry is created at the end of the list of entries of the block whose number is in the BN. The length of an entry is based on the number of information words allotted and the number of sets of X - Y quads allotted. Minimum length of an entry is 1. (The displacement to the next entry.) So, the total number of words allotted is: 1 + number of information words + 4 (number of sets of X - Y quads).

The EN in the current block is set to point to the new entry, and the LN is set to zero.

Insert Entry

INSE

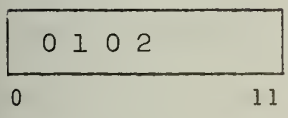


This instruction is the same as the APPE instruction, except that, rather than being added at the end of the list, the entry is inserted before the

entry pointed to by the EN. The EN is then set to point to this new entry, and the LN is set to zero.

Delete Entry

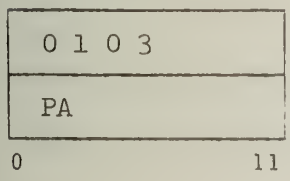
DELE



The entry whose number is contained in the EN is deleted. The EN is then set to point to entry immediately after the one deleted, unless, the entry deleted was the last entry in the list, then the EN is set to point to the entry immediately preceeding the deleted entry. If the entry deleted were the only entry in the block, the EN is set to zero.

Scan a Text Group

TSCAN PA



For this instruction, it is assumed that the top two levels of the stack contain the coordinates of a joystick hit, with the y coordinate on the top level. (These are not popped after instruction execution.)

The text group of the current entry is scanned for the line which, if displayed, would contain the coordinates given on the stack.

If the line is not found, the next instruction in line is executed, and the LN retains its original value.

If the line is found, the program branches to the address, PA, specified in the second word of the instruction, the LN is set to point

to the line, and the CN is set to point to the character within that line that contains the given coordinates. (For a description of the CN, see the RDCN instruction below.)

The method of scanning an entry is as follows:

It was shown in the beginning of this section that the coordinates of the n-th line in a text group are $(x_0 + DX, y - (n-1)hy + DY)$, where x_0, y_0 are the coordinates in the first two words of the text group, hy is the height of a character, and DX and DY are the offset words contained in the block header. Using this formula and the length of the current line to be examined, each line can be checked to see of the given coordinates fall anywhere within that line. That is, if the coordinates of the line under consideration are calculated to be (x_c, y_c), and the length in characters of this line is L, then this line is the referenced line if and only if:

$$x_c \leq x_{stack} < x_c + L\ hx$$

$$\text{and } y_c \leq y_{stack} < y_c + hy$$

In the following examples, consult the sample block pictured in FIGURE 9, and note that a character on the display screen has $hx = 12_{10}$ and $hy = 20_{10}$.

As an example of the calculation of coordinates and limits of a line, consider the fourth line in the second entry of FIGURE 9:

$$x_c = x_0 + DX = 320 + 0 = 320$$

$$y_c = y_0 - (n-1)hy + DY = 240 - (4-1)20 + 0 = 180$$

Thus, (320, 180) are the coordinates of the lower left hand corner of the leftmost character of line 4.

In order to reference this line, a joystick hit must be within the following limits:

SAMPLE BLOCK

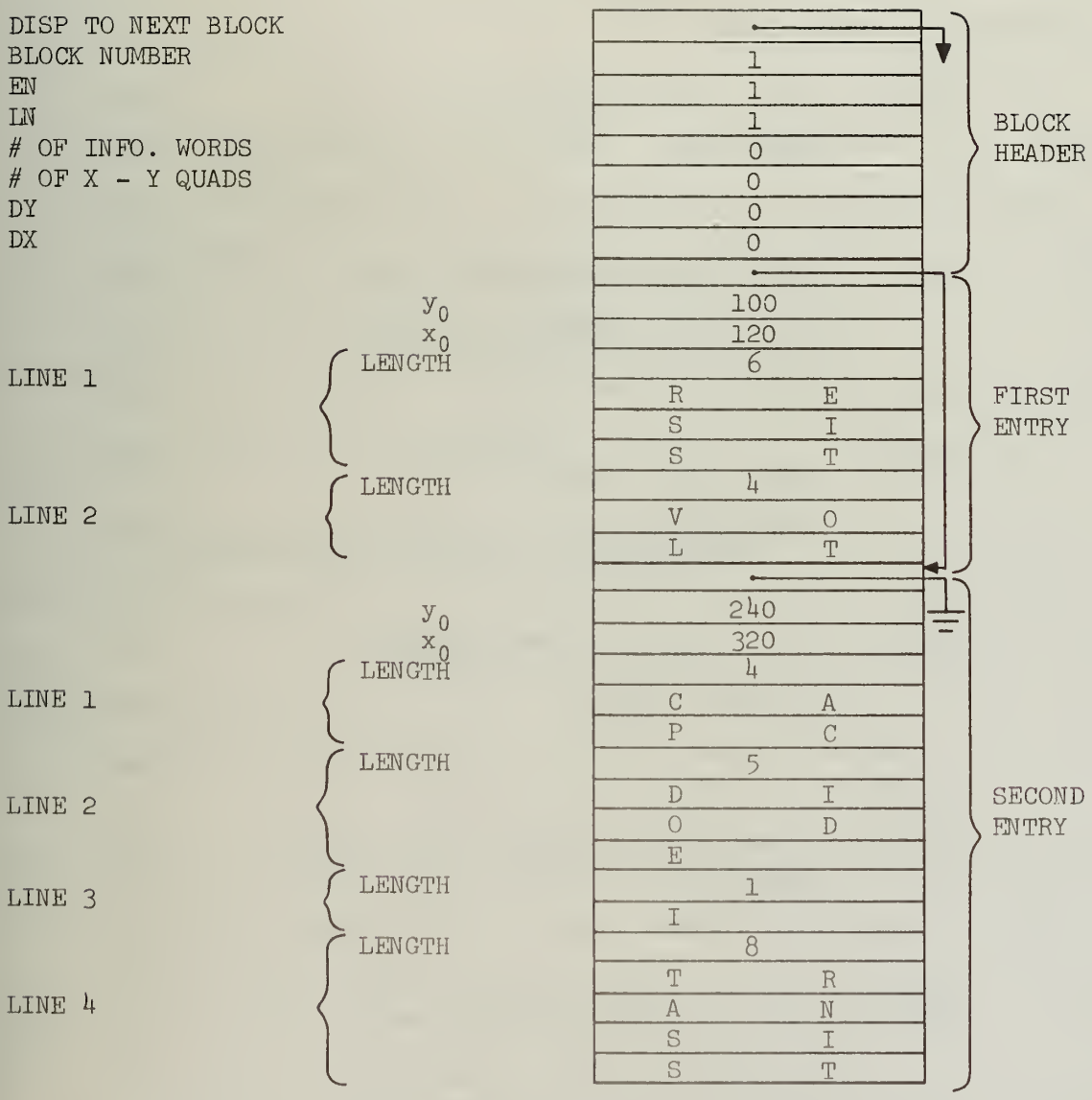


FIGURE 9

$$320 \leq x_{stack} < 320 + 8(12) = 416$$

$$\text{and } 180 \leq y_{stack} < 180 + 20 = 200$$

Some sample instructions,

a) Before: $y_{stack} = 79$, $x_{stack} = 150$ EN=1

TSCAN ADDR LN=1

CN=-

After: Line is not found, and the next EN=1

instruction in line is executed. LN=1

CN=-

b) Before: $y_{stack} = 87$, $x_{stack} = 150$ EN=1

TSCAN ADDR LN=1

CN=-

After: Line is found, and the program EN=1

branches to ADDR. LN=2

CN=3

c) Before: $y_{stack} = 195$, $x_{stack} = 330$ EN=1

TSCAN ADDR LN=1

CN=-

After: Line is not found, and the next EN=1

instruction in line is executed. LN=1

If the EN had been set to 2, the CN=-

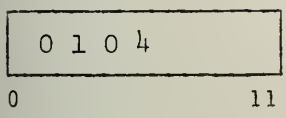
line would have been found, the

LN would have been set to 4, and

the CN to 1.

Read Character Number.

RDCN



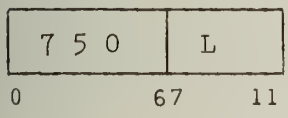
The CN is an internally kept word, which is set only by the TSCAN command. If the line referenced by the X - Y coordinates on the stack is found during the TSCAN command, the CN is set to point to the character within this line that contains the X - Y coordinates.

The RDCN command simply creates a new level on top of the stack and stores the contents of the CN in it.

4.4.3.1 Information Word and X - Y Quad Commands

Load X - Y Quad Word

LDQ L



Bits 7-11 of the instruction word are treated as a five bit positive integer, L. The coordinate pair stored in words L and L + 1 of the X - Y Quads section of the current entry is pushed onto the stack. (Word L is on the top level.)

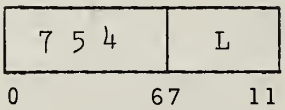
If, however, L = 0, the top of the stack is popped and the contents of this word is treated as a 12 bit positive integer, L'. Then, the coordinate pair stored in words L' and L' + 1 of the X - Y Quads section of the current entry is pushed onto the stack. (Word L' is on the top level.) This enables words numbered higher than 31 to be loaded.

Note: If an entry has n sets of X - Y Quads associated with it, then

it has 4n words in the X - Y Quad section, numbered from 1 to 4n.

Load Information Word

LDW L

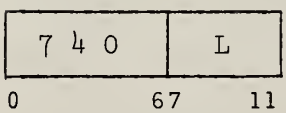


Bits 7-11 of the instruction word are treated as a 5 bit positive integer, L. The L-th word of the Information Word section of the current entry is pushed onto the stack.

If, however, L = 0, the top of the stack is popped and the contents of this word are treated as a 12 bit positive integer, L'. Then, the L'-th word of the Information Word section of the current entry is pushed onto the stack. This enables words numbered higher than 31 to be loaded.

Store X - Y Quad Word

STQ L



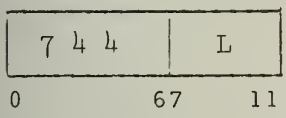
Bits 7-11 of the instruction word are treated as a 5 bit positive integer, . The coordinate pair contained in the top two levels of the stack is stored in words L and L + 1 of the X - Y Quads section of the current entry. (The top level is stored in word L.) The top two levels are then popped.

If, however, L = 0, the top of the stack is popped and the contents of this word are treated as a 12 bit positive integer, L'. Then the contents of the new top two levels of the stack are stored in words L' and L' + 1 of the X - Y Quads section of the current entry. These two words are then popped off the stack.

Note: If an entry has n sets of X - Y Quads associated with it, then it has 4n words in the X - Y Quad section, numbered from 1 to 4n.

Store Information Word

STW L

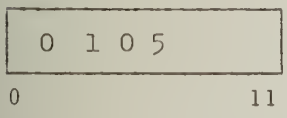


Bits 7-11 of the instruction word are treated as a 5 bit positive integer, L. The contents of the top level of the stack are stored in the L-th word of the Information Word section of the current entry. The top of the stack is then popped.

If, however, L = 0, the top of the stack is popped and the contents of this word are treated as a 12 bit positive integer, L'. Then, the contents of the new top level of the stack is stored in the L'-th word of the Information Word section of the current entry, and the stack is popped again.

Copy Information Word

COPYW



The contents of all of the information words of one entry are copied into the information words of another entry.

The entry which is to receive the data, E_{to}, has its file, block, and entry numbers contained in the FN, BN and EN, respectively. The entry which is providing the data, E_{from}, has its file, block and entry numbers in the second, first and top (zero-th) levels of the stack, respectively.

The copying is performed as follows:

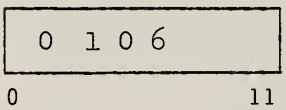
1) Both entry numbers are checked to ensure that an entry exists in the blocks specified.

2) The number of information words specified in the block headers in each of the designated blocks is checked to ensure that both entries have the same number of words allocated.

3) All of the information words of E_{from} are copied into the information words of E_{to}. (Note: The information words of E_{from} are not changed in any way.)

Copy X - Y Quads

COPY Q



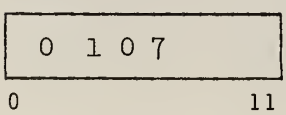
This instruction functions exactly the same as COPY W, except that, rather than all of the information words being copied, the entire X - Y quads section (all sets) of one entry is copied into the X - Y quads section of the other entry.

4.4.3.2 Text Group Commands

All Text group commands refer to the current line which is contained in the current entry, whose number is in the EN. The number of the current line is contained in the LN.

Set Line Number

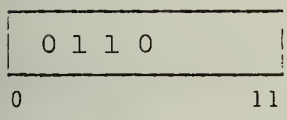
SETLN



The top level of the stack is examined. If the contents is not equal to -1, it is stored in the LN. If it is equal to -1, the number of the last line in the text group is stored in the LN.

Read Line Number

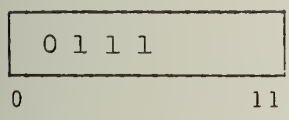
RDLN



A new level is created on the top of the stack, i.e., TP ← TP + 1, and the contents of the LN is stored in this level.

Append a Line to Text Group

APPL



The line in the TB beteeen P1 and P2 is appended to the entry at the end of the text group. This is done as follows:

1) If this is the first line in the text group, two words are allocated for the coordinates of the text group.

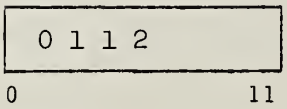
2) N words are allocated at the end of the text group, where:
$N = \left\lceil \frac{LENG}{2} \right\rceil + 1$, and LENG is the length in characters of the text string to be stored. LENG is contained in the TB header.

3) The length in characters of the text string is stored in the first word, and the text string is stored in 6 bit ASCII in the remaining N - 1 words. The LN is then set to point to the new line (See FIGURE 10). After the line is stored in the entry, the TB is

initialized, i.e., P1 = P2 = 1, LENG = 0.

Insert Line into Text Group

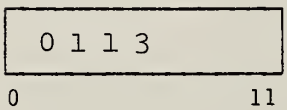
INSL



The action is the same as APPL, except that rather than being appended to the end of the Text Group, the line is inserted before the line pointed to by the LN. The LN is then set to point to the inserted line (See FIGURE 10), and the TB is initialized.

Delete a Line from the Text Group

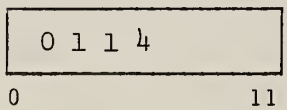
DELL



The line pointed to by the LN is deleted. If this is the only line in the text group, the X - Y coordinates are also deleted, and the LN is set to 0. Otherwise, the LN is set to point to the line immediately following the deleted line. See FIGURE 10. (If the last line in the text group is deleted the LN is set to point to the next to last line.)

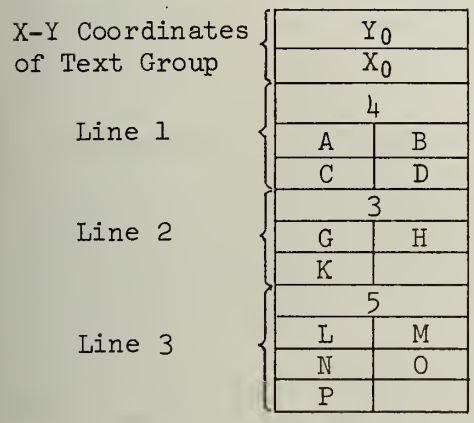
Change a Line in the Text Group

CHGL



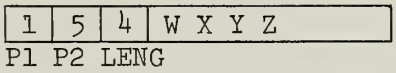
The line pointed to by the LN is deleted and the line in the TB between P1 and P2 is stored in its place. (The Text Group is appropriately

Before:
Sample Text Group:

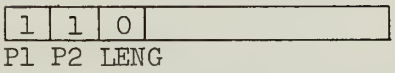


LN = 2

Before:
Contents of TB:

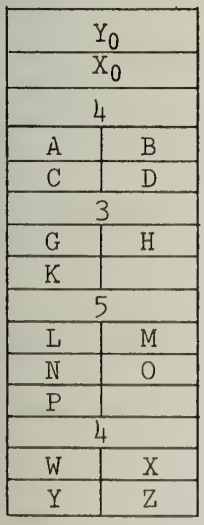


After APPL, INSL and CHGL:
Contents of TB:



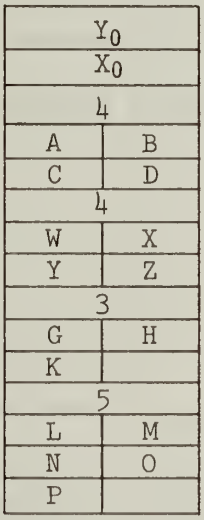
DELL does not affect the TB.

After APPL Inst:



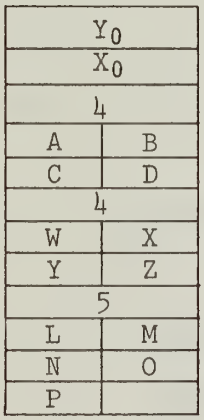
LN = 4

After INSL Inst:



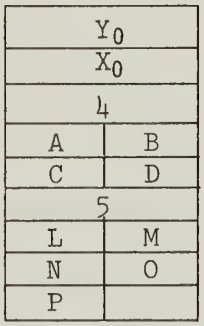
LN = 2

After CHGL Inst:



LN = 2

After DELL Inst:



LN = 2

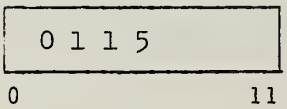
Sample Text Group and Instructions

FIGURE 10

expanded or contracted.) See FIGURE 10. After the line is stored in the entry, the TB is initialized.

Load Line

LOADL



The line pointed to by the LN is loaded into the TB. (It is not deleted from the Text Group.) Whatever was previously in the TB is destroyed. The LN remains the same.

The TB header is set as follows:

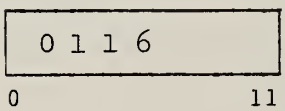
P1 = 1

P2 = length of line in characters + 1

LENG = length of line in characters

Load x - y Coordinates

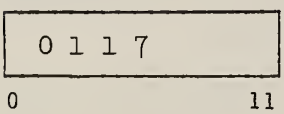
LDXYT



The x coordinate word of the Text Group in the current entry is pushed onto the stack, and then the y coordinate word is pushed onto the stack.

Store x - y Coordinates

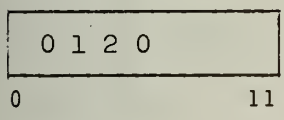
STXYT



The top level of the stack is popped and stored in the y coordinate word of the Text Group in the current entry. Then, the next level is popped and stored in the x coordinate word.

Load Length of Text Line

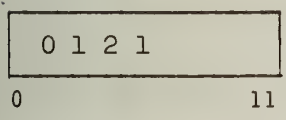
LDLEN



The length of the current line is pushed onto the stack. (This length is the number of characters in the line.)

Copy Text Group

COPYT



The contents of the Text Group of one entry is copied into another entry.

The entry which is to receive the data, E_{to}, has its file, block, and entry numbers contained in the FN, BN, and EN, respectively. The entry which is providing the data, E_{from}, has its file, block and entry numbers in the second, first and top levels of the stack, respectively.

The copying is performed as follows:

1) Both entry numbers are checked to ensure that an entry exists in the blocks specified.
2) E_{to} is allocated N words, where N is the total number of words in the Text Group of E_{from}. (The block containing E_{to} is expanded appropriately.)

3) The contents of the Text Group of E_{from} is copied into the newly allocated words of E_{to}. (Note that the Text Group of E_{from} is not altered in any way.)

4.5 Input/Output Commands

4.5.1 Input/Output Control

In order to get some understanding of the type of control and program structure required in a model description system, a very simple system will be presented and explained.

FIGURE 11 depicts a sample screen which the programmer of the model description system has created to enable the user to construct models.

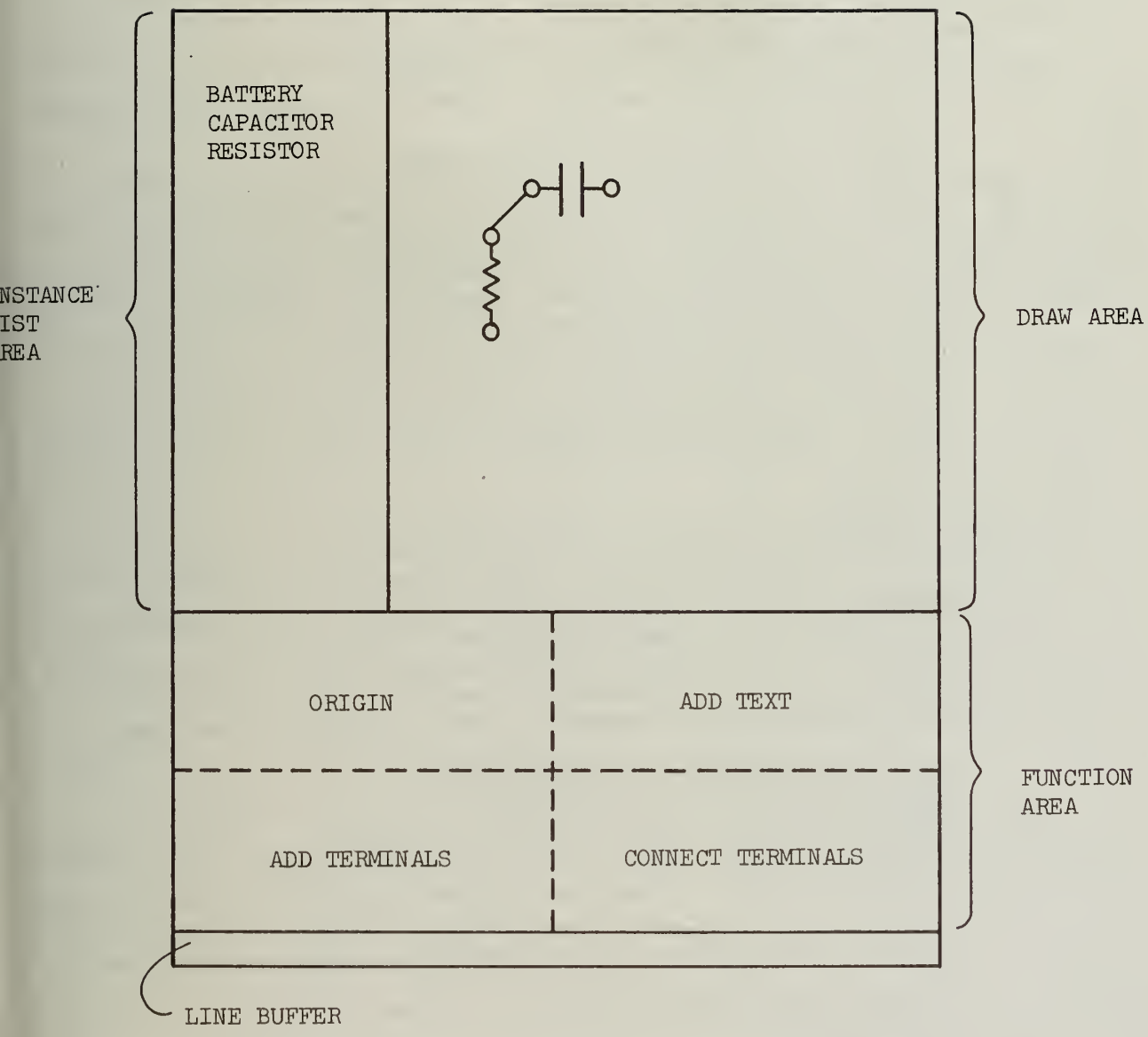
The screen is divided into three basic areas:

Area 1: This section of the screen contains a list of items (instances) which the user has previously constructed. An item may be used in the current picture by "joysticking" the name of the item. The item is then incorporated into the picture and displayed (in Area 2) at the current cursor position. (The current cursor position is described below in the ORIGIN function.) Note that in the partially constructed picture in Area 2 of FIGURE 11, the user had previously joysticked RESISTOR and CAPACITOR.

Area 2: The current picture is constructed in this area of the screen. It is assumed that, unless the system is performing a function that requires a joystick hit, any hit in Area 2 will result in a line, drawn from the current cursor position to the joystick hit.

Area 4: This section contains four functions which aid in model construction:

1) ORIGIN - This function establishes the current cursor position. If, after this function has been chosen, the next joystick hit is



SAMPLE SCREEN

FIGURE 11

in Area 2, this hit is defined as the current cursor position. The current cursor position is used as an assumed set of coordinates in most of the other functions. After the current cursor position has been set, the function is terminated.

2) ADD TEXT - After choosing this function, the user types a text line into the line buffer. When a carriage return is typed, the line is incorporated into the picture, and displayed below the previously typed line. The first line is displayed at the current cursor position. This function remains in effect until another function is chosen.

3) ADD TERMINAL - A terminal is the point at which a picture may be connected to other pictures. When this function is invoked, a terminal is incorporated into the current picture, and displayed at the current cursor position. After the terminal has been constructed, the function is terminated.

4) CONNECT TERMINALS - After choosing this function, the user joysticks two terminals. The connection is noted in the data structure, and a line is drawn between the two terminals. This function remains in effect until another function is chosen.

From the superficial description of this simple system, it should be obvious that a model description system mainly consists of functions which assist the user in the construction of his model.

These functions are basically two types:

1) Immediate: This type of function can be performed simply by choosing the function. No additional data is required. For instance, incorporating the items listed in Area 1 into the current picture is an immediate function.

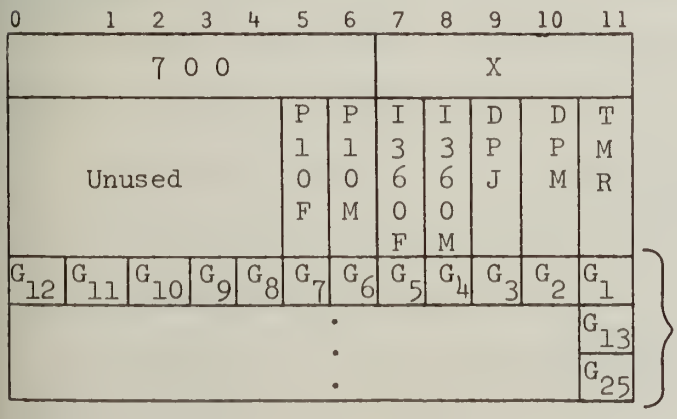
2) Delayed: This type of function requires additional input after the function has been chosen. The functions which connect terminals, add text, or reposition the current cursor position are delayed functions.

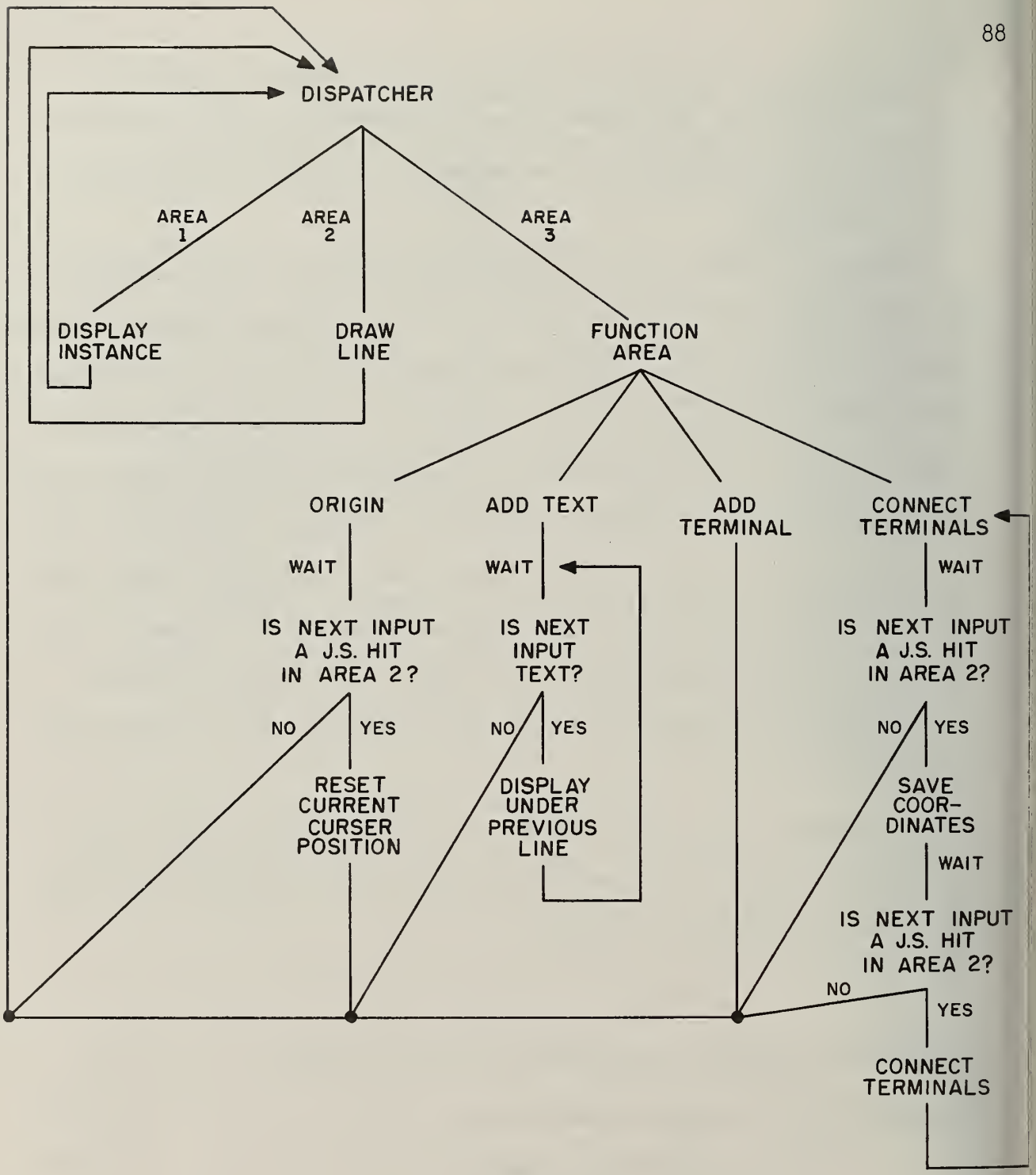
Examining FIGURE 12, it is evident that the flow of control depends on the type of function. When an immediate function has been performed, control should be returned to the dispatching program, so that the next input may be directed to the proper function. However, the delayed functions must retain control to accept and process the additional data that they require. In the case of delayed functions, the type of input also affects the flow of control. Most functions are expecting a particular type of input, and wish to retain control only if this input is received. For example, the "ADD TEXT" function only wishes control when a text string is received. If a joystick hit is input, control should be returned to the dispatching program.

As a result of the above considerations, the WAIT instruction was designated to enable the procedure waiting for input to specify the conditions for its retention of control.

Wait

WAIT {List of devices from which input will be accepted.}





FLOW OF CONTROL

FIGURE 12

A model description program is "input-driven," that is, in order to build picture files of models, a program must continually receive commands or data from the user. The WAIT command enables a procedure to specify the devices from which it will accept input, and then suspends execution of this procedure until some input is received. When input is received:

If the device which is sending input is one of those specified in the WAIT instruction, the procedure which issued the WAIT regains control at the instruction after the WAIT, and the number of the device sending the input is pushed onto the stack.

If the device which is sending input is not one of those specified in the WAIT instruction, a normal RETURN (See discussion of subroutines in section 4.6) is performed for the procedure, P, which issued the WAIT, and the procedure which originally called procedure P is given control at the instruction after the call. The device number is not pushed onto the stack, and the program which has just gained control must issue its own WAIT instruction to examine the input. (Note: If the routine issuing the WAIT is the Main routine, i.e., it was not called by any routine, an error message is given if input is received which wasn't in the WAIT instruction.)

There are seven devices which can provide input:

Device number 1 - Timer, TMR: Associated with each console is an interval timer word. After this word has been set to some value, it is incremented by one every 1/60 of a second. When it reaches zero, an interrupt occurs. Thus, "requesting input" from the timer is simply a request to be notified if the timer has reached zero. The timer is initially disabled, and is set with the SETTMR instruction. (See section 4.5.4) After reaching zero, an interrupt occurs and the timer

is again considered to be disabled until another SETTMR instruction is issued. The interval timer may be used, for example, to "log out" a user if his console has remained unused for a long period of time.

The console, or display processor, may send input either in message form (text string), or joystick form (coordinate pair). These are considered to be two different types of input and are requested separately.

Device number 2 - Display Processor Message, DPM - A message is a text string of 72 characters or less created at the display console Keyboard.

Device number 3 - Display Processor Joystick Hit, DPJ - A joystick hit is the selection of a particular point on the screen, and is described by a coordinate pair. The joystick is commonly used to draw lines, or to select items on the screen.

Devices 4 to 7 refer to the remote storage devices, PDP/10, and IBM 360/75. Input may be received from these devices in either message or file form. These two forms are requested separately for each device:

Device number 4 - IBM 360 Message, I360M - The message is a text string of 72 characters or less sent from the IBM 360/75.

Device number 5 - IBM 360 File, I360F - The file is a group of 1024_{10} words or less sent from the IBM 360/75.

Device number 6 - PDP/10 Message, P10M.

Device number 7 - PDP/10 File, P10F.

In the WAIT instruction, bits 5-11 of the second instruction word are used to specify the devices from which the program will accept input. The bit associated with each device is pictured in the second instruction word. (These bits may be set by listing the device mnemonics, in any order, to the right of the WAIT.) Since joystick hits in different areas of the screen may imply different functions, it is possible to further qualify

the acceptance of a joystick hit. By means of a grid, the screen may be sectioned into different areas. (A grid logically divides part or all of the screen into a number of rectangular segments, which are numbered starting with one, from left to right, top to bottom. Any number of grids may be defined, but only one at a time may be declared as the current grid. For a complete discussion of grids see Section 4.5.2.2.) Thus, when specifying input devices, rather than accept all joystick hits, the program may list the screen segments of the current grid, within which, the joystick hit must be contained. (This is done by listing the screen numbers, enclosed in parentheses, immediately after the DPJ mnemonic. If no screen numbers are listed, it is assumed that any joystick hit is acceptable.)

Instruction word 3 and those following are used to indicate the designated segment numbers. Each bit in these words represents a screen segment number, and the bits are numbered from right to left as pictured in the instruction. Instruction words are allocated until the largest designated screen number, N, is reached. Therefore, the total number of words needed to list the segment numbers is $\left\lceil \frac{N}{12} \right\rceil$. This number is kept in bits 7-11 of the first instruction word. (Note: If the DPJ bit is set, and bits 7-11 of the first instruction word are zero, all joystick hits are accepted.)

Some sample instructions follow:

a) WAIT DPJ, DPM, P1OF assembles as:

7 0 0 0
0 1 0 6

Control is returned to the original procedure if any input is received from the console, or if a file is received from the PDP/10.

Otherwise, the program which called this procedure regains control.

b) WAIT DPJ (3, 5, 12, 15), TMR assembles as:

7 0 0 2
0 0 0 5
4 0 2 4
0 0 0 4

Control is returned to the original procedure if the timer has interrupted, or if there were a joystick hit in screen segments 3, 5, 12, or 15 of the current grid, when the instruction is executed. (Note: If there were a hit in any other screen segment, control is not returned to the original procedure.) Otherwise the program which called this procedure regains control.

c) Consider the following code, and assume that the WAIT command in procedure B has just been executed.

```
A:    PROC
      SETGRD G1
      CALL B
      WAIT DPJ(3, 5, 7), DPM
      TSTBR 3, DPJS              See if input were a j.s. hit.
      POP 1                      It was a text string. Pop device
      RCVDPM                     number, and receive message.
        .
        .
        .
DPJS: RCVDPJ                     Input was a j.s. hit.
        .
        .
        .
```

```
B:    PROC
      SETGRD G2
      WAIT I360F, DPJ(1, 2, 3)
      TSTBR 5, FILE              See if input were a 360 file.
      POP 1                      No, it was a j.s. hit.
      RCVDPJ                     Pop device number, and get input.
        .
        .
        .
FILE: RCV360F                    Input was a file.
        .
        .
        .
```

1) If input were a file from the IBM 360, procedure B would retain control and would continue execution at the instruction after the WAIT.

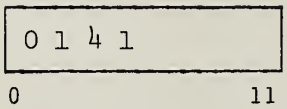
2) If input were a joystick hit, it would be examined to determine whether it were in segments 1, 2, or 3 of grid G2. If not, a return from procedure B would automatically be executed, and procedure A would then gain control at its WAIT instruction. If the hit were in segment 3, 5, or 7 of grid G1, A would retain control and continue execution at the instruction after the WAIT.

Thus, the WAIT command allows control to be passed to the procedure which is equipped to handle the current input. It does not, however, clear the interrupt flag of the device, nor provide the input to the procedure. When a procedure has returned from a WAIT, and wishes to process the input, it must first determine which device provided the input. The device number is placed on the stack by the WAIT command, and is usually examined with the TSTBR command. After determining the device, the procedure must then clear the device interrupt flag and accept or reject the input by issuing one of the RCV commands or the FLUSH command. (These are presented in the next few sections.) Note that a device is "locked up" and may not send

more input until a RCV or FLUSH command is issued.

Flush

FLUSH



The device interrupt flag of the current device is cleared, and the input from this device is destroyed. The current device is the device whose number was placed on the stack by the most recently executed WAIT command.

4.5.2 Console Commands

A Display Terminal (or Console) consists of the following: (See FIGURE 13)

1) Display Screen - This is used for picture construction and display. A small dot, called a cursor, is continually displayed on the screen, and pictures are constructed by positioning the cursor, by means of the joystick, into a desired position. When the cursor is positioned at the chosen location, one of the twelve function buttons is depressed, and the coordinates of the cursor are sent to the model description program. (See FIGURE 15 for a layout of the screen coordinates.) The action taken by the program is then determined by the position of the joystick hit, and the significance assigned to the particular function button employed.
2) Console Keyboard - The keyboard is used to construct text lines to be displayed or sent to the model description program. At the bottom of the display screen, there is a display screen line buffer. When a character is typed at the keyboard, it is displayed

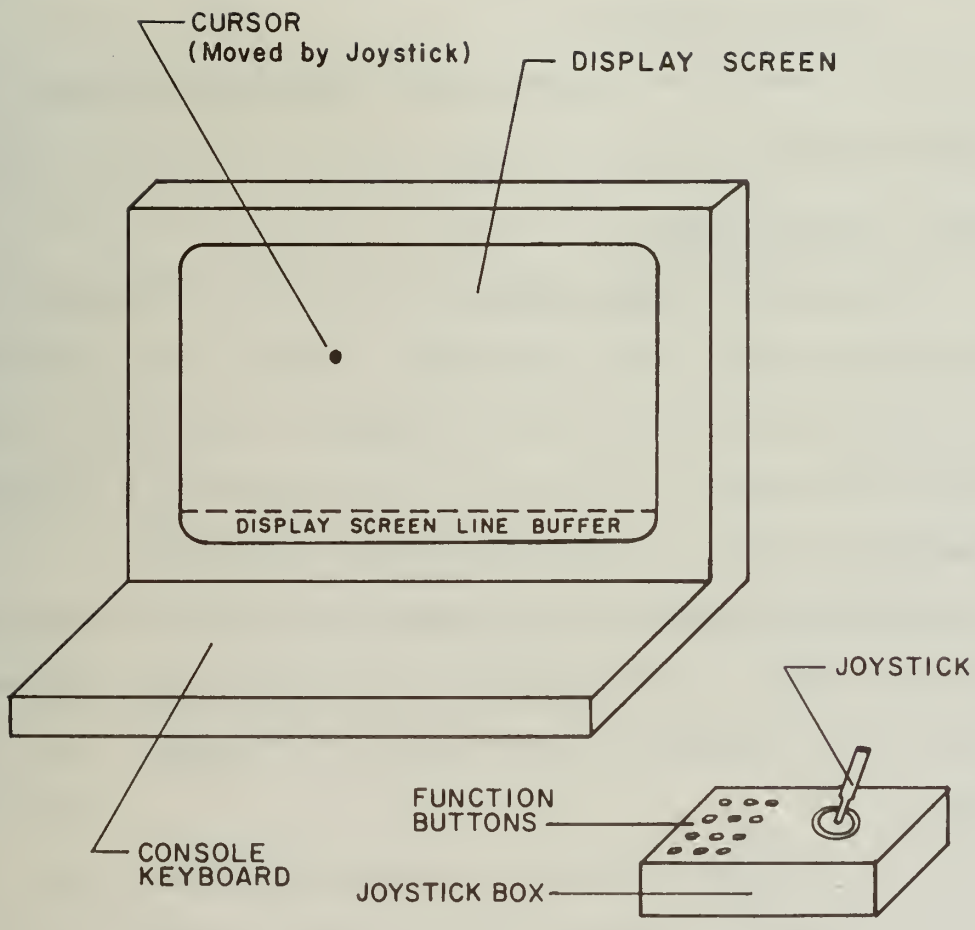

CURSOR
(Moved by Joystick)
DISPLAY SCREEN
DISPLAY SCREEN LINE BUFFER
JOYSTICK
FUNCTION
BUTTONS
CONSOLE
KEYBOARD
JOYSTICK BOX

FIGURE 13

in the line buffer immediately to the right of the previous character typed. When a carriage return is typed the line in the buffer is sent to the model description program and the buffer is cleared.

The following three sections contain commands pertaining to the Display Terminal.

4.5.2.1 Display Commands

The interpreter language provides instructions to display either a file, block, entry or part of an entry. However, since entries contain all of the information that can be displayed, "display block" simply means that all the entries in a block are to be displayed, and "display file" means that all the blocks in a file are to be displayed.

As mentioned previously, each entry can contain any combination of the three data types: Information Words, X - Y Quads, and a Text Group. However, only X - Y Quads and Text Groups can be displayed. This is done as follows:

a) X - Y Quads: A single X - Y Quad contains four words. The first two words are assumed to be the Y and X coordinates, respectively, of one endpoint and the second two words are assumed to be the Y and X coordinates of the other endpoint. Thus, if an X - Y Quad containing k, l, m, n in words 1 to 4 is to be displayed, a line is constructed on the screen from (l, k) to (n, m).

b) Text Group: If a Text Group is to be displayed, the first line is displayed with its leftmost character starting at the X - Y coordinates of the Text Group, and each successive line is displayed directly under the previous one.

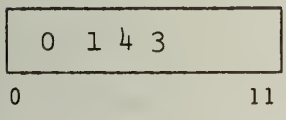
If just the n-th line in a text group is to be displayed, it is

displayed with its leftmost character starting at (x_0 + DX, y_0 - (n-1)hy + DY), where (x_0, y_0) are the coordinates specified at the beginning of the Text Group, hy is the height of a character, and DX, DY are the offset words contained in the block header. (For additional explanation, see FIGURES 6, 7, and 8, section 2, and the introduction to section 4.4.)

Finally, note that a picture remains on the display screen until it is erased. Thus, two consecutive "display file" commands would be displayed on top of each other, unless an erase were inserted between them.

Display File

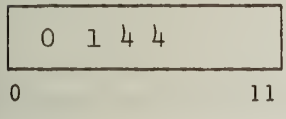
DISPF



The file whose number is contained in the FN is displayed on the display screen.

Display Block

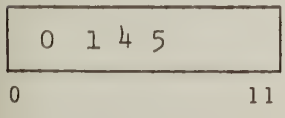
DISPB



The block whose number is contained in the BN is displayed on the display screen.

Display Entry

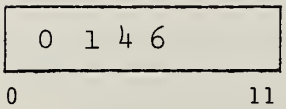
DISPE



The entry whose number is contained in the EN is displayed on the display screen. When an entry is displayed, all of its sets of X - Y Quads and its Text Group are displayed.

Display X - Y Quad

DISPQ

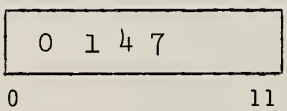


The X - Y Quad whose number is contained in the top level of the stack is displayed on the display screen. The stack is then popped.

Note: If an entry contains n X - Y Quads, and the m-th is to be displayed, the top of the stack should contain the number m. If $m \leq 0$ or $m > n$, an error is given.

Display Text Line

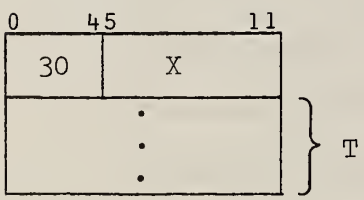
DISPL



The line in the Text Group of the current entry whose number is contained in the LN is displayed on the display screen.

Display Text String

DISPTXT T



It is assumed that the top two levels of the stack contain an X - Y coordinate pair. (The Y coordinate is assumed to be in the top level.)

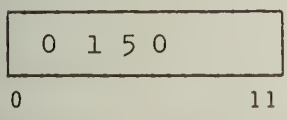
The text string, T, is displayed on the screen at the coordinates specified on the stack. (The stack is not popped.)

Bits 5 to 11 of the first instruction word are used to hold a count of the number of words needed in the instruction to represent the text string, T. (This number is determined by the assembly.)

Note that if no text string is specified in the instruction, bits 5 to 11 are set to zero. Then, the contents of the Text Buffer, between P1 and P2, is displayed on the screen at the coordinates specified on the stack.

Load Console Line Buffer

LDCNSLB



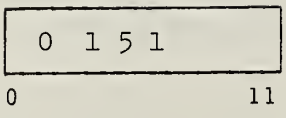
As mentioned above, each console has a line buffer used for constructing text lines from the console keyboard.

This instruction loads the display line buffer with the line in the TB between P1 and P2. The TB is then initialized, i.e., P1 = P2 = 1, LENG = 0.

Once the line is in the display line buffer, the user may use the console keyboard to revise the line.

Erase

ERASE



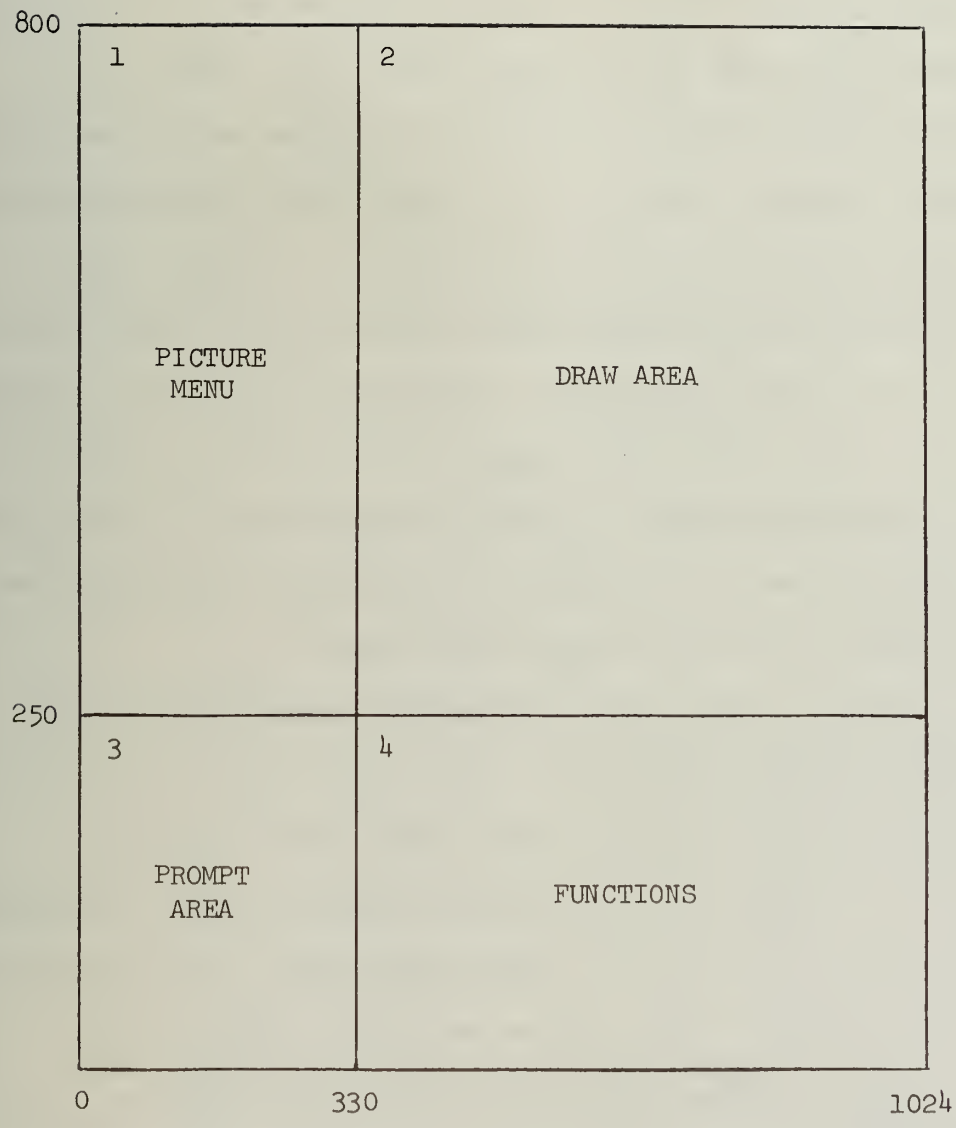
This instruction erases the entire display screen.

4.5.2.2 Grids

Grids enable the display screen to be subdivided into logical segments. The programmer may then associate various functions with different segments, and perform separate operations depending on which segment the user "joysticked."

As a simple example, consider FIGURE 14. A grid has been described which divides the screen into four logical areas. If this were to be used in a simple model description system, segment one could contain a list of previously constructed pictures, segment two could be the area used for constructing the current picture, segment three could contain suggestions concerning what function to perform next, and segment four could contain a list of the functions provided. So, the user could "joystick" in segment one to request that a picture be displayed, in segment two to draw lines, and segment four to initiate a function.

The following pseudo-operation is used to define a grid.



A: GRID X (0, 330, 1024), Y (0, 250, 800)

FIGURE 14

GRID X (X_1, ..., X_n), Y (Y_1, ..., Y_n)

The x_i and y_i are decimal numbers, such that $0 \leq x_i \leq 1024_{10}$ and $0 \leq y_i \leq 800_{10}$, and $x_i < x_{i+1}$, $y_i < y_{i+1}$. (The ranges for the x_i and y_i are based on the size of the display screen. See FIGURE 15.)

This instruction defines a grid which subdivides the display screen into logical segments. The x_1, ..., x_n in the GRID instruction represent vertical lines which divide the screen into n - 1 sections. The y_i, ..., y_m represent horizontal lines which divide the screen into m - 1 segments. The inside of the grid is defined in the horizontal direction by x_1 and x_n, and in the vertical direction by y_1 and y_m. Inside the grid, the rectangular segments, formed by the intersection of the x_i and y_i, are numbered from left to right, top to bottom, starting with one. Anything outside the grid is assigned the number 0.

As an example, compare the following GRID instruction with its corresponding grid in FIGURE 15:

```
FIG15:    GRID X (256, 480, 736), Y (224, 320, 480, 608)
```

Also note in FIGURES 15 and 16 that a grid may or may not encompass the entire screen, and that it is sometimes helpful to define grids within grids. The instructions used to form the grids in FIGURE 16 are:

```
OUTER:    GRID X (0, 480, 1024), Y (0, 416, 800)
INNER:    GRID X (480, 736, 1024) (0, 224, 416)
```

Since this instruction is a pseudo-operation, it doesn't get assembled into the program, and, as a result, may be placed anywhere in the program. It must, however, have a label so that the grid definition can be referenced at assembly time, and stored in the SETGRD instruction.

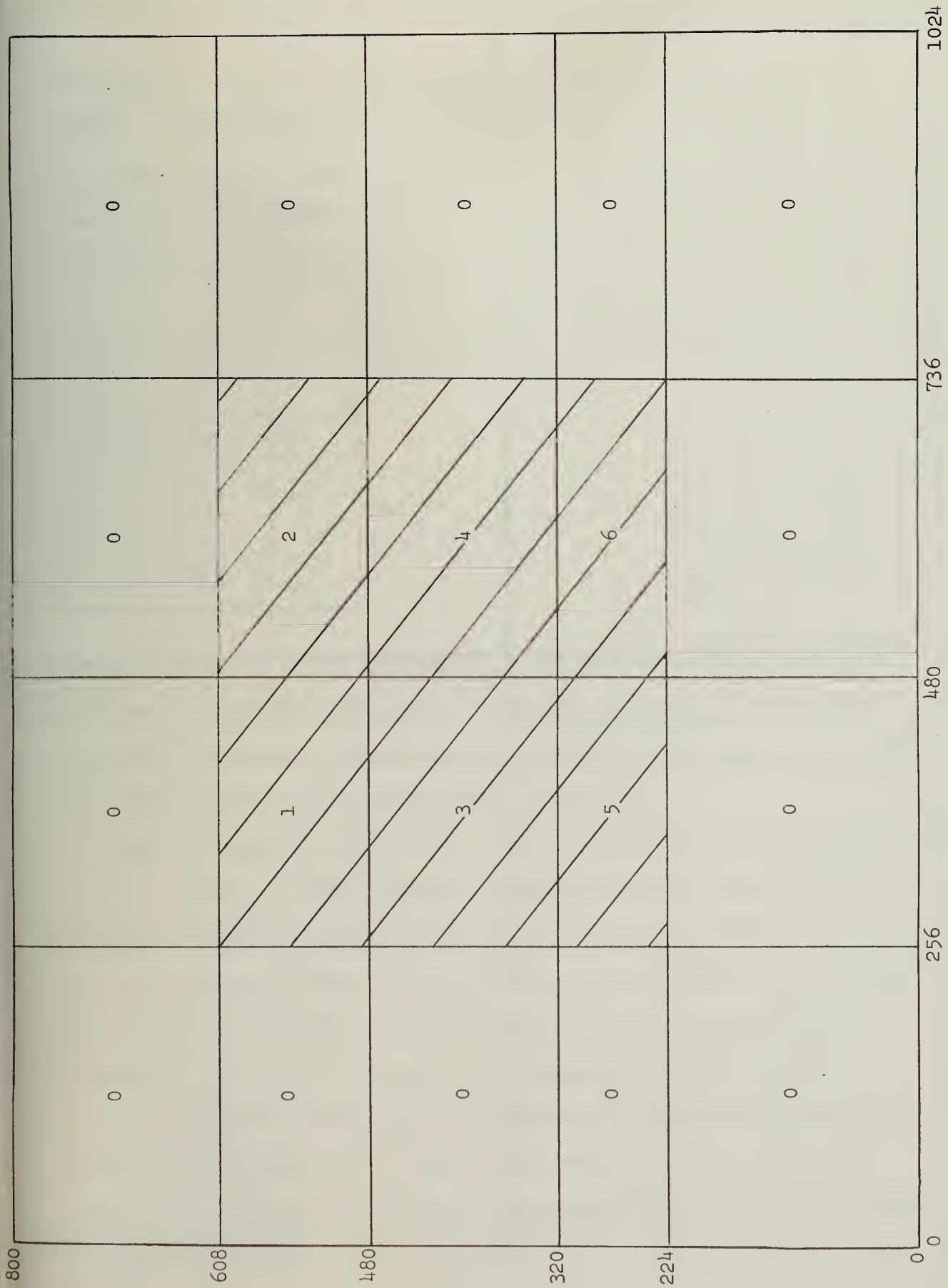


FIGURE 15

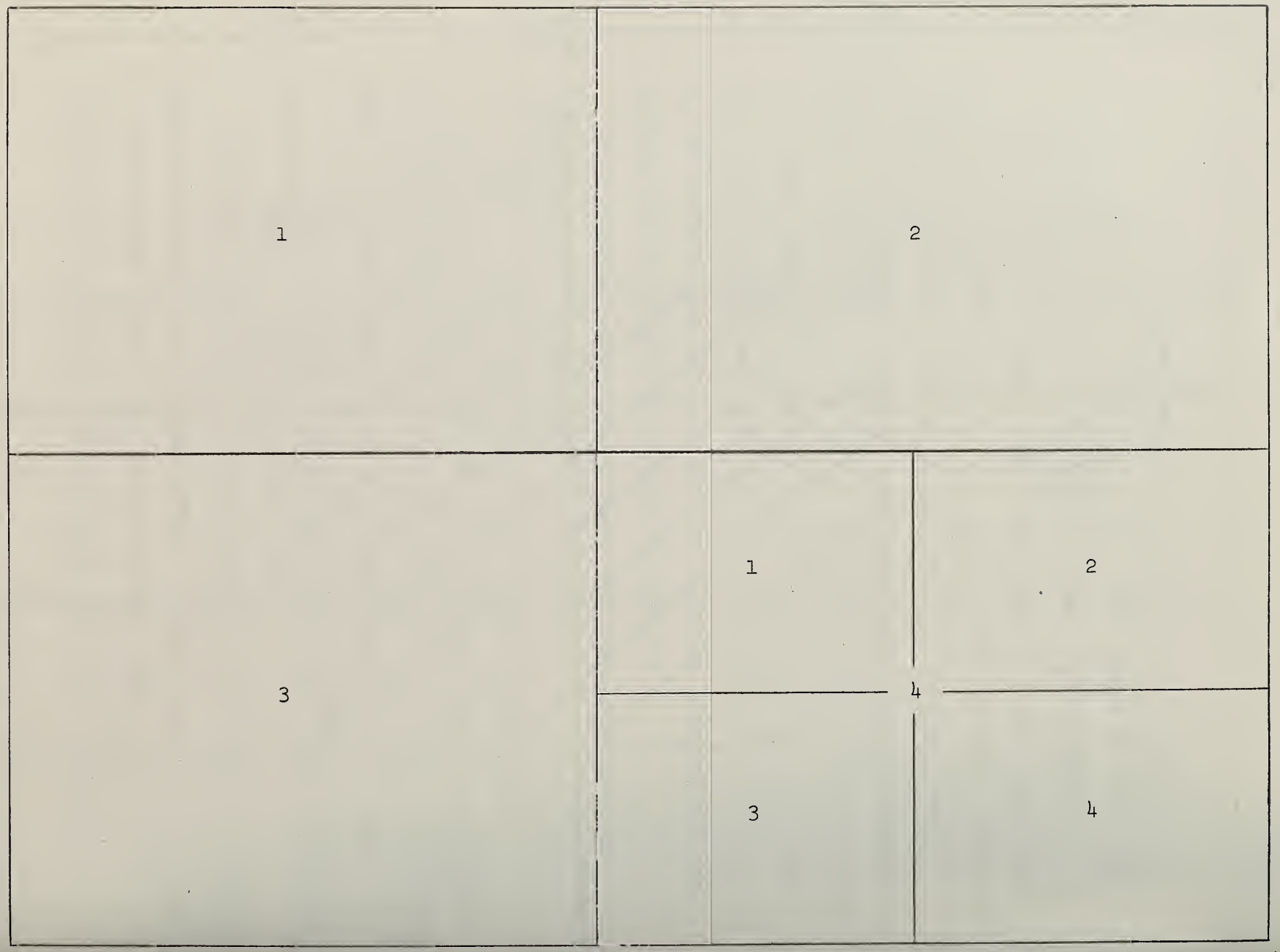
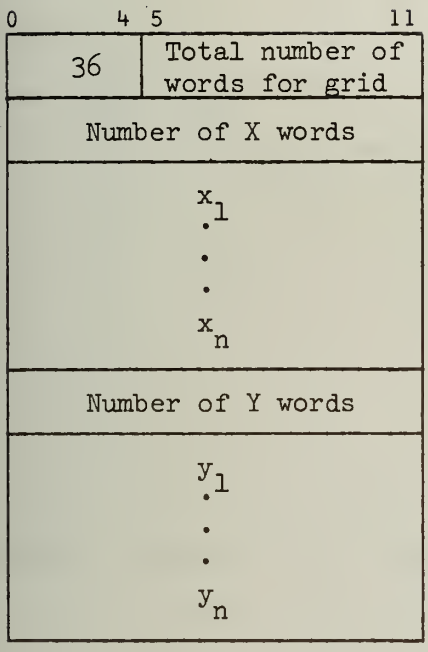


FIGURE 16

Set Grid

SETGRD Label of grid



The grid specified in the instruction is declared to be the current grid (for use in the GRDNUM instruction). Within a procedure, only one grid may be the current grid, and a grid designated as the current grid, remains the current grid until the next SETGRD instruction.

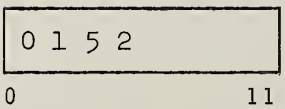
When a procedure passes control to another procedure through a subroutine call, the called procedure initially assumes the current grid, G, of the calling procedure. The called routine may then use the SETGRD instruction to designate another grid, G', as the current grid. However, when control is returned to the original calling routine, grid G is automatically re-established as the current grid. (See discussion in Section 4.6.)

During assembly, when a grid is referenced by the SETGRD instruction, the assembler determines the correct grid, and stores it in the SETGRD instruction, starting with the second instruction word, as follows: (Refer to GRID instruction.)

Word two contains the number of x coordinates specified. The next n words contain the x coordinates, starting with x_1, stored one per word. The number of y coordinates is then stored, followed by the m y coordinates, one per word, starting with y_1. Bits 5-11 of the first instruction word contain the total number of words needed to store the grid, n+m+2.

Determine Grid Number

GRDNUM



This instruction assumes that the coordinates of a joystick hit are on the top two levels of the stack (Y coordinate is on the top level).

These coordinates are examined, but not popped, to determine which segment of the current grid was referenced. The grid segment number is then pushed onto the stack. If the hit were not inside the grid, a zero is pushed onto the stack.

For example, consider the grid in FIGURE 15.

If the coordinates were:

a) Y=546, X=608, a two would be pushed onto the stack.

b) Y=128, X=384, a zero would be pushed onto the stack because the hit is not inside the grid.

The BRTBL instruction may be used with grids to branch to a function when the grid section has been determined. For instance,

SETGRD FIG15	Declare the grid in FIGURE 15 to be the current grid.
WAIT DPJ	Wait for any joystick hit
GRDNUM	Determine grid section
BRTBL FCNS	Branch to functions corresponding to each of the six sections.
FLUSH	Flush the joystick hit, because it was not in the grid. (A zero was on the top of the stack.)

.
.
.
.

FIG 8: GRID X(256, 480, 736), Y(224, 320, 480, 608) Grid definition.

FCNS: ADRTBL F1, F2, F3, F4, F5, F6 This is an assembler pseudo-operation which provides the definition of a branch table.

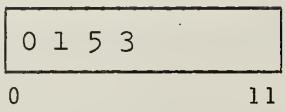
An important point to remember about grids is that they are a logical, not physical, subdivision of the screen. When a SETGRD instruction is issued, only an internal record is made of the current grid. No lines appear on the screen. If the user is to be made aware of the current grid through which his joystick hits will be filtered, the programmer of the model description system must actually construct the lines which define the grid, and display them on the screen.

4.5.2.3 Input Commands

The following commands are executed after a WAIT instruction. When data is sent from the console to the program, these commands accept the input and free the device.

Receive Display Processor Message

RCVDPM



When this instruction is executed, the text string, which has been sent from the console, is placed in the Text Buffer, and the DPM interrupt flag is cleared.

The TB header is set as follows:

P1 = 1

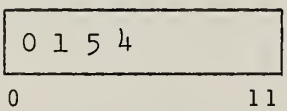
P2 = Length of text + 1

LENG = Length of text

The console is then free to send input again.

Receive Display Processor Joystick Hit

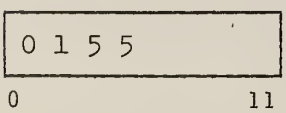
RCVDPJ



When this instruction is executed, the coordinates of the joystick hit are pushed on the stack (the Y coordinate is stored in the top level), and the interrupt flag is cleared. The console is then free to send input again.

Load Function Button Number

LDFBN

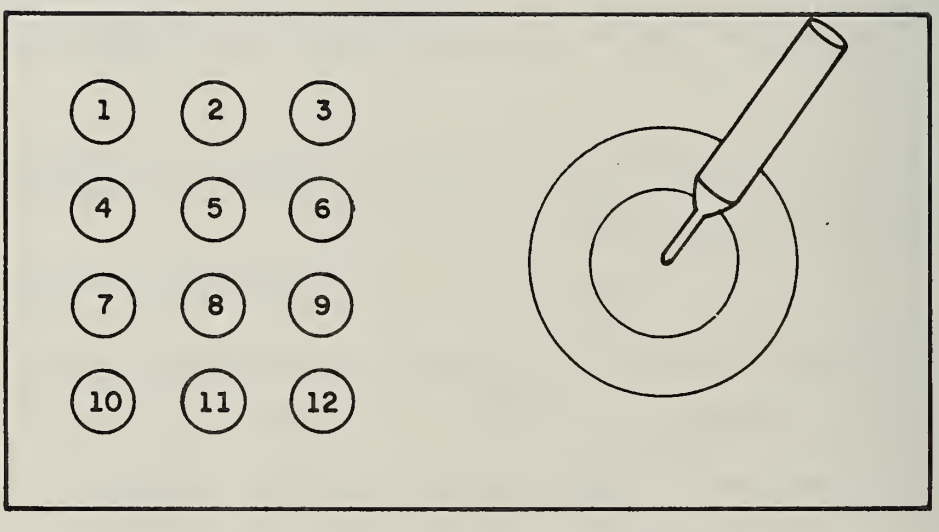


As mentioned above, the joystick has a function button box associated with it, which contains 12 buttons. (See FIGURE 17 for the numbering of the buttons.) A joystick hit is registered if any one of these buttons is depressed. They may be employed by the programmer to determine the action taken for a particular joystick hit. (For example, a button may be used to request screen regeneration, or to apply horizontal or vertical line drawing constraints on the joystick hits which follow.) Thus, associated with every joystick hit are an x-y coordinate pair and a function button number.

This instruction pushes onto the stack the number of the function button used in the current joystick hit.

The following section of code requests and accepts any type of input from the display console:

```
        WAIT  DPM, DPJ      Wait for either joystick or message
                            input from console.

        TSTBR #2, DPMSG     See if device number on the stack is
                            DPM number.

        RCVDPJ              No, so it must be a DPJ.

        LDFBN               Accept joystick hit, then get
                            function button number.

           .
           .
           .
           .

DPMSG:  RCVDPM              Input was a message.  Accept it into
                            TB.

           .
           .
           .
```



JOYSTICK AND FUNCTION BUTTONS

FIGURE 17

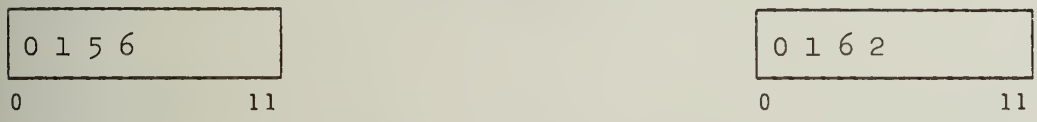
4.5.3 Storage Commands

4.5.3.1 Remote Communication Commands

The remote computers, the IBM 360/75 and the PDP/10, are primarily used for analysis of models and long term storage of user files. The following commands enable files or single line messages to be transmitted or received from the remote computers. Files are used to transfer pictures, and single line messages.

Send A Single-Line Message

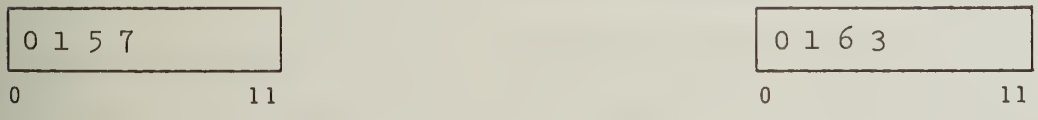
SEND360M or SENDP10M



The text line in the TB pointed to by P1 and P2 is sent to the program running in the remote computer specified. The TB is then initialized, i.e., P1=P2=1 and LENG=0.

Send A File

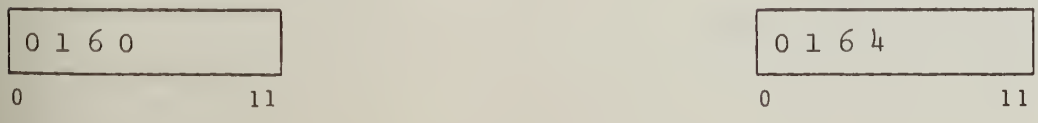
SEND360F or SENDP10F



The file pointed to by the FN is sent to the program running in the remote computer specified.

Receive A Single-Line Message

RCV360M or RCVP10M



When this instruction is executed, the text string, which has been sent from the remote computer specified, is stored in the TB.

The TB header is set as follows:

P1 = 1

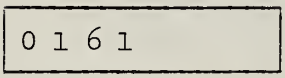
P2 = Length of text + 1

LENG = Length of Text

The interrupt flag is cleared. So, the remote computer is free to send another message.

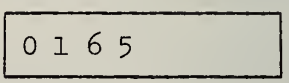
Receive A File

RCV360F



0 11

RCVP10F



0 11

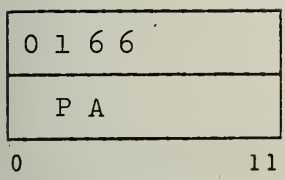
The file sent from the remote computer specified is stored in the file whose number is contained in the FN. Whatever was previously in this file is destroyed. The interrupt flag is cleared.

4.5.3.2 Local Storage Commands

The Local Storage Commands handle files that are kept locally on a disk. A file on disk is referenced by its name, which can be at most six alphanumeric characters, the first of which must be alphabetic, its extension number, which must be between 0 and 77_8, and a protection number, which may be any number between 0 and 7777_8. When specifying a file on disk for reference by a command, the name is placed in the TB between P1 and P2, and the extension and protection numbers are stored in the zero-th and first levels of the stack.

Load A File

FLOAD PA

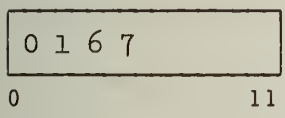


The disk file specified is loaded into the existing file whose number is in the FN, then the top two levels of the stack (containing the extension and protection numbers) are popped.

If the file specified is not on disk, the program branches to the address listed in the second instruction word.

Save A File

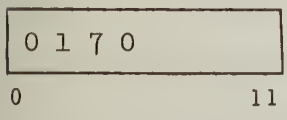
FSAVE



The file whose number is contained in the FN is saved on the disk under the name, extension and protection numbers specified. Any file previously saved under the same name, extension and protection numbers is destroyed.

Delete A File

FDEL

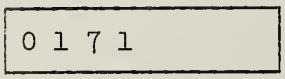


The file specified is deleted from the disk.

4.5.4 Miscellaneous Devices

Set Timer

SETTMR



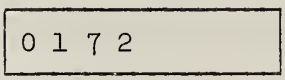
This instruction is used to set the interval timer described above in the WAIT instruction.

The contents of the top level of the stack is stored in the interval timer word. The stack is then popped.

Note that since the timer is incremented every 1/60 of a second, a negative number is normally stored in the timer. (The number that can be stored in the timer ranges from -2048_{10} to $+2047_{10}$.)

Receive Timer

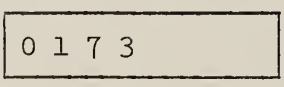
RCVTMR



This instruction simply clears the device interrupt flag for the timer. At this point, the timer is disabled and no further interrupts will occur until the timer is reset by the SETTMR command.

Print Teteype Message

TTYPRT



The contents of the TB between P1 and P2 is printed on the operator's

teletype.

This instruction may be used to request operator intervention, such as, loading a tape.

4.6 Subroutines

A section of instructions which is to be executed at a number of different points in a program may be grouped into a subroutine. Then, rather than including the entire section every time it is to be executed, a branch (call) is made to the subroutine, and control is transferred to the designated sequence of instructions. Upon completion of the task, control is then returned to the original routine at the instruction immediately following the call. In this particular language, there are two types of subroutines: external and internal. This division into two types was primarily created to enable the efficient handling of large programs. It promotes modularization of independent tasks and decreases the amount of storage required to reference a program address.

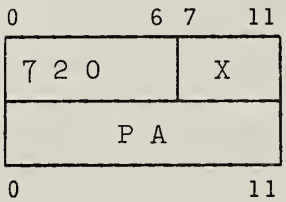
When the problem of addressing was first being considered, it was decided that, due to the small amount of memory, program addresses must be referenced using only one word. As a result, a maximum of 4096 words, or 32 sections (pages) of 128 words each, may be addressed. To accomodate larger programs, it was decided to associate a name with each group of 32 pages. A program begins execution in one of the groups. This group is considered to be the current group. All address references are assumed to refer to instructions within the current group, and, as a result, require only a page and offset. To reference addresses outside of the current group, a transfer (call) is made to the group containing the desired address. In this case, a name, page and offset must be provided,

and this new group becomes the current group until a return is made to the original group, or a call is made to another group.

Any instruction not contained within a declared procedure (subroutine) is assumed to be contained in the main procedure. The main procedure is considered to be a group with the name "MAIN". When another procedure is defined, the name of the group which is to contain this procedure must be provided.

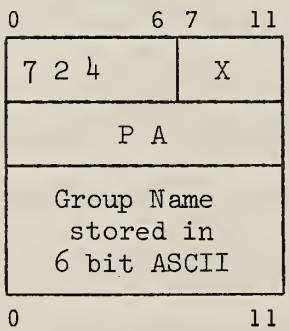
Call (Internal) Procedure

CALL Procedure Name (Parameters)



Call External Procedure

CALLE Procedure Name (Parameters), Group Name



CALL: The procedure called is within the current 32 page group. A transfer is made to the procedure, and execution begins at the program address specified.

CALLE: The procedure called is outside the current 32 page group. The group whose name is specified in instruction words 3, 4, 5 becomes

the current group, and execution begins in this group at the program address specified.

It is important to note that, normally, external procedures are only used in large programs or when referencing separately assembled procedures. Since the CALLE requires three extra words of storage, external procedures should be used judiciously.

Bits 7-11 of the first instruction word in both types of call contain the number of parameter words associated with the call. Parameters are discussed below.

The type of call employed depends entirely on the locations of the procedure to be called and the calling procedure. If they are in different groups, a CALLE must be used, if not, a CALL is used. So, in the course of a program, a procecure may be called at one time via a CALLE, and at another via a CALL.

When a procedure is called, either by CALL or CALLE, the linkage between procedures is handled internally by the interpreter program. (A description of this is provided in Appendix C.) However, there are some important points which should be remembered, when control is passed from one procedure to another.

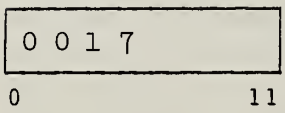
1. If a grid, G, were declared current in the calling procedure, A, it continues to be the current grid in the called procedure, B, until a another grid is declared current. However, when control is returned to procedure A, grid G is re-established as the current grid.

2. When a procedure is called, the address, A, of the top level of the stack is saved, and the called procedure may not directly reference or modify local variables below this level. If the called procedure wishes to reference or modify any stack variable below this level, the variable

must either have been declared as a global variable, or it must be passed as a parameter by the calling routine. (Parameters are discussed below.) The called procedure may increase the stack above its current level by declaring local variables, or pushing data onto the stack, but when control is returned to the calling procedure, the stack is returned to its original address, A, and anything above this level is lost. Again, parameters or global variables would be used to pass information back to the calling procedure.

Return

RETURN



When the return command is executed within a called procedure, control is returned to the calling procedure at the instruction immediately following the call.

The group name of the calling procedure becomes the current name again. The stack level is set to the level just before the call, and the grid which was current just before the call is re-established as the current grid.

4.6.1 Parameters

If the called routine needs to reference the variables of the calling routine, the variables must be passed as parameters by the calling routine. These parameters, which are usually local variables, are specified by listing them, enclosed in parentheses, immediately after the call command.

When a call instruction is being assembled, the parameters are examined, and an appropriate LOADA is then generated after the normal call instruction

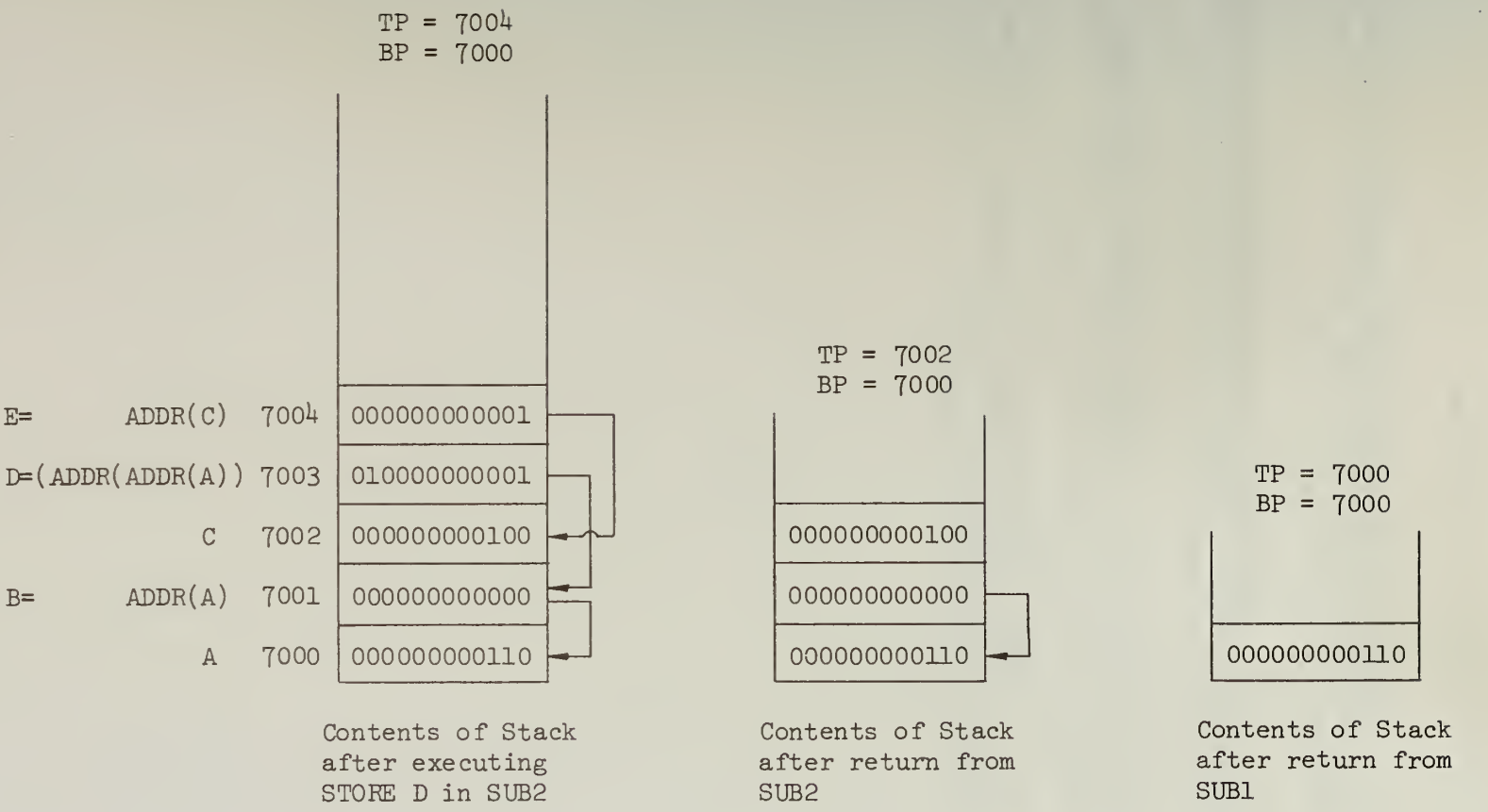
for each of the parameters. The total number, X, of words generated for the parameters is stored in bits 7-11 of the first word of the call instruction. Then, during execution of the call instruction, after the stack level of the calling procedure has been saved, the X instructions after the call are executed. These instructions load the addresses of the variables required onto the called procedure's stack. (The called procedure may then reference these variables indirectly through the addresses provided.) Finally, the return address is set to point to the X+1st word after the call, and control is transferred to the called procedure.

An example is provided in FIGURES 18 and 19. The procedures in the example were designed simply to demonstrate the mechanics of subroutines and parameter passing. It would be helpful to review section 3.2.1.1 on dynamic variable addressing before studying the example.

Address	Contents (OCTAL)	Instructions	
--	--	LOCAL A	
0000	3607	SETGRD G1	The grid instructions have
0001	0003		been inserted only to show
0002	0000		how the grid words are set
0003	0062		in the subroutine stack.
0004	0144		(See Appendix C.)
0005	0002		
0006	0024		
0007	0036		
0010	0402	LOADA #2	
0011	0601	STORE A	
0012	7201	CALL SUB1(A)	
0013	0016		
0014	0400	LOADA A	Generated by Assembler.
0015	0017	RETURN	
--	--	G1: GRID X(0,50,100), Y(20,30)	
--	--	END	
		SUB1: PROC(B), MAIN	
		LOCAL C	
0016	0404	LOADA #4	
0017	0601	STORE C	
0020	7242	CALLE SUB2(B,C)	
0021	0000		
0022	2325		
0023	0262		
0024	4040		
0025	0441	LOADA B	Generated by Assembler.
0026	0401	LOADA C	Note that Address of B
0027	0017	RETURN	must be indirect.
--	--	END	
--	--	SUB2: PROC(D,E), EXT	
0000	3606	SETGRD G2	
0001	0002		
0002	0062		
0003	0113		
0004	0002		
0005	0000		
0006	0144		
0007	0241	LOAD D	Assembler sets indirect
0010	1641	ADD E	reference bit in object
0011	0642	STORE D	code.
0012	0017	RETURN	
--	--	G2: GRID X(50,75), Y(0,100)	
--	--	END	

Sample Program Demonstrating Subroutines and Parameters

FIGURE 18



Contents of Variable Stack during Program in Figure 18

FIGURE 19

List of References

[1] Gear, C. W. An Interactive Graphic Modeling System, Department of Computer Science Report No. 318, University of Illinois, Urbana, Illinois 61801 (April 1969)

[2] Gear, C. W., et al The Simulation and Modeling System - A Snapshot View, Department of Computer Science File No. 824, University of Illinois, Urbana, Illinois 61801 (February 1970)

[3] Michel, M. J. and Koch, J. A. GRASS: Terminal User's Guide, Department of Computer Science Report No. 467, University of Illinois, Urbana, Illinois 61801 (August 1971)

[4] Michel, M. J. GRASS: System Software Description, Department of Computer Science Report No. 468, University of Illinois, Urbana, Illinois 61801 (August 1971)

APPENDIX

A. Opcode Description

In FIGURE 20, a decoding tree has been provided, and the bits in a sample instruction word have been labeled for reference in the ensuing discussion.

If bits 0-4, LLLLL, of the instruction word are zero, the instruction has no operand stored in the first instruction word, and the opcode is the octal value of 00000MMNNNNN. This allows a maximum of 128 "no-operand" instructions. The opcode assignments were distributed over the four major instruction groups: Arithmetic and Logical, Text Buffer, Data Structure and Input/Output. (See TABLE 1.) Since approximately 58% of the possible opcodes have been assigned, a large amount of room remains for expansion.

If LLLLL is not equal to zero, the instruction contains either a five or seven bit operand. In the case of a seven bit operand, LLLLL is the opcode, and the remaining bits are used to contain the operand. For a five bit operand, two additional bits, MM, are used to distinguish between the four possible instructions which have the same LLLLL. Thus, the opcode is LLLLLMM, and, excepting branch instructions, the remaining five bits are used to contain the operand.

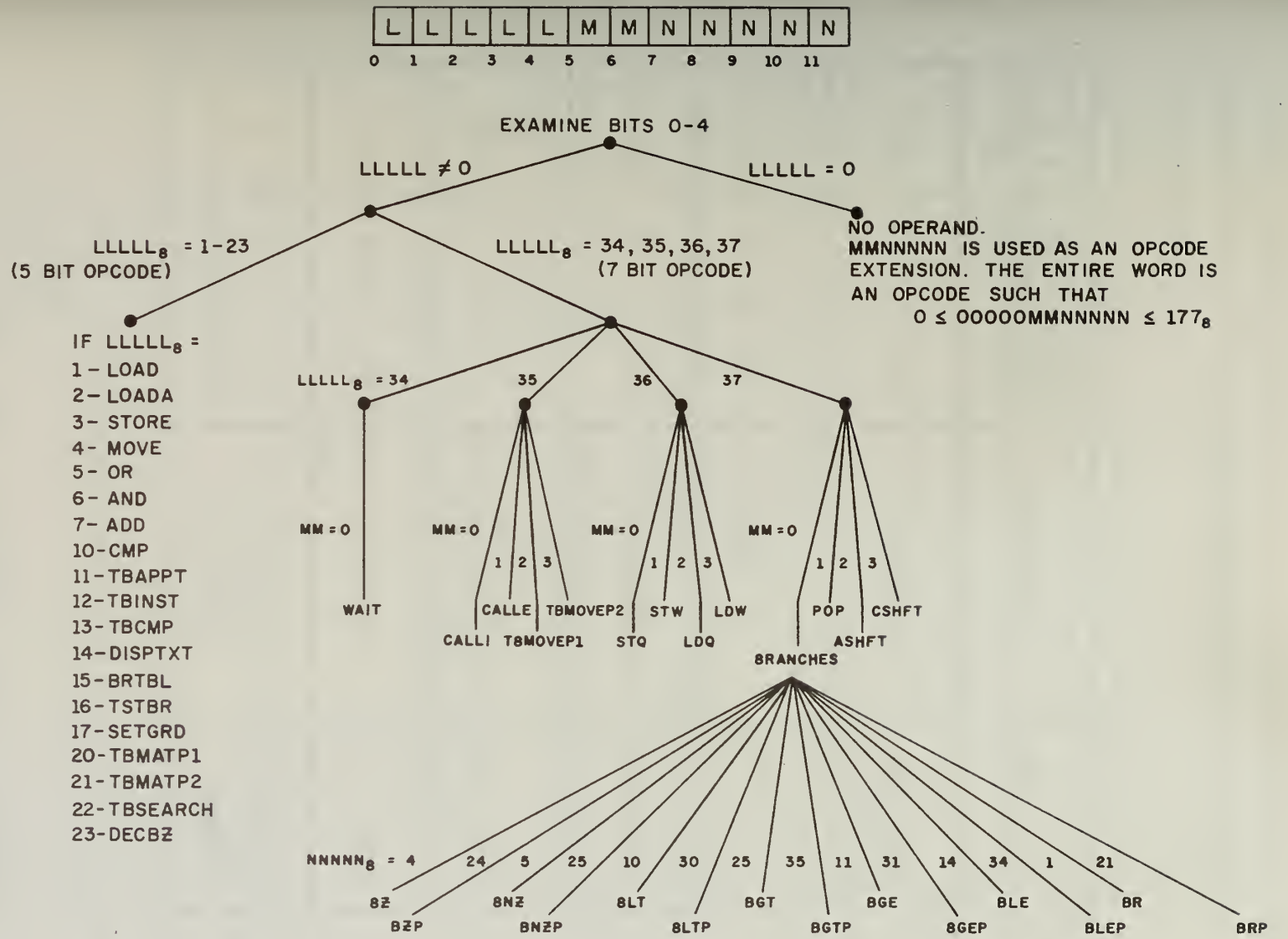

L L L L L M M N N N N N
0 1 2 3 4 5 6 7 8 9 10 11
EXAMINE BITS 0-4
LLLLL ≠ 0
LLLLL = 0
NO OPERAND.
MMNNNNN IS USED AS AN OPCODE EXTENSION. THE ENTIRE WORD IS AN OPCODE SUCH THAT
0 ≤ OOOOOMMNNNNN ≤ 177₈
LLLLL₈ = 1-23
(5 BIT OPCODE)
LLLLL₈ = 34, 35, 36, 37
(7 BIT OPCODE)
IF LLLLL₈ =
1 - LOAD
2 - LOADA
3 - STORE
4 - MOVE
5 - OR
6 - AND
7 - ADD
10 - CMP
11 - TBAPPT
12 - TBINST
13 - TBCMP
14 - DISPTXT
15 - BRTBL
16 - TSTBR
17 - SETGRD
20 - TBMATP1
21 - TBMATP2
22 - TBSEARCH
23 - DECBZ
LLLLL₈ = 34
35
36
37
MM = 0
WAIT
1 2 3
CALLI
CALLE
TBMOVEP1
TBMOVEP2
STQ
LDQ
STW
LDW
POP
ASHFT
CSHFT
BRANCHES
NNNNN₈ = 4
24
5
25
10
30
25
35
11
31
14
34
1
21
BZ
BZP
BNZ
BNZP
BLT
BLTP
BGT
BGTP
BGE
BGEP
BLE
BLEP
BR
BRP

FIGURE 20

OPCODE DISTRIBUTION

TYPE	INST. GROUP	OPCODE RANGE	TOTAL # OF INST.	# OF ASSIGNED INST.	# OF REMAINING INST.
NO OPERAND	ARITH, LOG & CONTROL	0000-0017	16	4	12
	TEXT BUFFER	0020-0057	32	14	18
	DATA STRUCT	0060-0137	48	31	17
	I/O	0140-0177	32	26	6
	TOTAL	0000-0177	128	75	53
OPERAND	5 BIT & 7 BIT OPERANDS	0200-7740 (5 and 7 bit)	31 (7 bit	23 (7 bit)	8 (7 bit)

TABLE 1

Using the method described below, the branch instructions employ the remaining five bits as an opcode extension to specify the type of conditional branch.

Branch Instructions - (LLLLL=37, MM=0) - It was noticed that the opcodes for the five PDP/8 conditional instructions (SMA, SPA, SZA, SNA, SKP), which will be used to execute the interpreter branch instructions, differ only in four bit places. So, it is possible to encode bits NNNNN in an interpreter branch instruction with the proper 4-bit pattern to indicate which PDP/8 conditional instructions are needed to implement that particular branch instruction. This allows all branch instructions to be executed using the same routine.

So, NNNNN is assigned as follows: The leftmost N is assigned a 0 if the stack is to be popped after the instruction, and a 1 if not. The remaining NNNN are used to hold the appropriate bit pattern (TABLES 2 and 3 should clarify this).

Therefore, in order to determine which PDP/8 instruction to execute, it is simply necessary to perform the following operations on the opcode of the given branch instruction:

a) Clear all bits but NNNNN

b) Determine if stack is to be popped after instruction, and then clear leftmost bit.

c) Shift left 3.

d) Add 7400. This is the opcode for the required PDP/8 conditional instruction.

Referring to TABLE 1 again, there are a total of 31 possible opcodes for "seven bit operand" instructions. This seems reasonable because the language is primarily a zero-address (stack) language, and, thus, there

should be considerably fewer one-address instructions than zero-address. Twenty-three of the total number have been assigned, leaving eight for expansion. So, approximately 30% of the "operand" opcodes are still unassigned. (Note: These figures are for seven bit operand instructions. Four "five bit operand" instructions can be created from each "seven bit operand" instruction.)

PDP/8 Conditional Instructions

Inst.	Opcode	Bit Pattern	Significant 4 bit pattern
SKP	7410	111100001000	0001
SZA	7440	111100100000	0100
SNA	7450	111100101000	0101
SMA	7500	111101000000	1000
SPA	7510	111101001000	1001

TABLE 2

Graphics Language Branch Instructions

Inst.	Corresponding PDP/8 Inst(s).	NNNNN	(Octal)	Full Opcode
BZ	SZA	00100	(4)	7604
BZP	SZA	10100	(24)	7624
BNZ	SNA	00101	(5)	7605
BNZP	SNA	10101	(25)	7625
BMI	SMA	01000	(10)	7610
BMIP	SMA	11000	(30)	7630
BPL	SPA SNA	01101	(25)	7625
BPLP	SPA SNA	11101	(35)	7635
BPZ	SPA	01001	(11)	7611
BPZP	SPA	11001	(31)	7631
BMZ	SMA SZA	01100	(14)	7614
BMZP	SMA SZA	11100	(34)	7634
BR	SKP	00001	(1)	7601
BRP	SKP	10001	(21)	7621

TABLE 3

B. Instruction List

TYPE	ARITHMETIC AND LOGICAL	OPCODE	CONTROL AND SUBROUTINES	OPCODE	TEXT BUFFER	OPCODE	DATA STRUCTURE	OPCODE	INPUT/OUTPUT	OPCODE
NO OPERAND	NOOP	0000	RETURN	0017	TBINIT	0020	SETFN	0060	FLUSH	0141
	NEG	0001			TBPUSHC	0021	RDFN	0061	DISPF	0143
	COM	0002			TBINSC	0022	INITF	0062	DISPB	0144
					TBDELC	0023	SETBN	0065	DISPE	0145
					TBCHGC	0024	RDBN	0066	DISPQ	0146
					TBAPPC	0025	INITB	0067	DISPL	0147
					TBDELT	0026	DELB	0070	LDCNSLB	0150
					TBCNV	0027	LDDY	0072	ERASE	0151
					TBCMPRS	0030	LDDX	0073	GRDNUM	0152
					TBLDP1	0031	STDY	0074	RCVDPM	0153
					TBLDP2	0032	STDX	0075	RCVDPJ	0154
					TBLDLENG	0033	SETEN	0076	LDFBN	0155
					TBSTP1	0034	RDEN	0077	SEND360M	0156
					TBSTP2	0035	APPE	0100	SEND360F	0157
							INSE	0101	RCV360M	0160
							DELE	0102	RCV360F	0161
							TSCAN	0103	SENDP10M	0162
							RDCN	0104	SENDP10F	0163
							COPYW	0105	RCVP10M	0164
							COPYQ	0106	RCVP10F	0165
							SETLN	0107	FLOAD	0166
							RDLN	0110	FSAVE	0167
							APPL	0111	FDEL	0170
							INSL	0112	SETTMR	0171
							DELL	0113	RCVTMR	0172
							CHGL	0114	TTYPRT	0173
							LOADL	0115		
							LDXYT	0116		
							STXYT	0117		
							LDLEN	0120		
							COPYT	0121		
FIVE BIT OPERAND	POP L	7640	CALL	7200	TBMOVEP1 L	7300	LDQ L	7500	WAIT DEVICES	7000
	ASHFT L	7700	CALLE	7240	TBMOVEP2 L	7340	LDW L	7540		
	CSHFT L	7740	BZ	7604			STQ L	7400		
			BZP	7624			STW L	7440		
			BNZ	7605						
			BNZP	7625						
			BM1	7610						
			BM1P	7630						
			BPL	7615						
			BPLP	7635						
			BPZ	7611						
			BPZP	7631						
			BMZ	7614						
			BMZP	7634						
			BR	7601						
			BRP	7621						
SEVEN BIT OPERAND	LOAD DVA	0200	BRTBL $PA_1 \ldots PA_n$	3200	TBAPPT T	2200			DISPTXT T	3000
	LOADA DVA	0400	TSTBR L,PA	3400	TBINST T	2400			SETGRD LABEL	3600
	STORE DVA	0600	DVBNZ DVA,PA	4600	TBCMP T	2600				
	MOVE DVA	1000			TBMATP1 T,PA	4000				
	OR DVA	1200			TBMATP2 T,PA	4200				
	AND DVA	1400			TBSEARCH T,PA	4400				
	ADD DVA	1600								
	CMP DVA	2000								

C. Subroutine Linkage

When a procedure, A, calls another procedure, B, certain facts about both procedures must be saved so that procedure B may function independently, and, upon its completion, A may be restored to its original state. So, when procedure A calls procedure B, the information already stored for procedure A is adjusted to reflect the call, and the following words are allocated for procedure B on the subroutine stack:

1) Outer Program word - is stored with the location in the subroutine stack of the beginning of the stored information for procedure A.
2) Return Address word - will contain the return address of a procedure called by B. Initially, it is set to 0. The corresponding word in the stored information for procedure A is set to the address at which procedure A will continue when procedure B terminates.
3) Group Name Pointer word - is stored with the location in the subroutine stack of the group name of procedure B.
4) Stack Level word - will contain the address of the top level of the stack when procedure B calls another procedure. Initially, it is set to 0. The corresponding word in the stored information for procedure A contains the address of the top level of the stack when B was called.
5) Current Grid Pointer word - is set to contain the location in the subroutine stack of the current grid for procedure B. Initially, this is set to point to the current grid of procedure A.

If procedure B were called externally, the next three words contain the Group Name.

Finally, if procedure B sets a grid to be the current grid, it is stored in the subroutine stack, and the grid pointer word for B

is set to point to it. (Each procedure may only have one current grid at a time. So if a procedure issues more than one SETGRD instruction, the old grid is deleted before the new one is stored.

When procedure B issues a return, all of the words allocated on the subroutine stack during the call to B are deleted, the variable stack is set to the address stored in the stack level word of procedure A, and A continues execution at the address stored in its return address word. In FIGURE 21, the movement of the subroutine stack is shown for the program presented in FIGURES 18 and 19. (Only one area is allocated for use by both the subroutine stack and the variable stack. So, the stacks start at different ends of the area and proceed toward the middle. The base of the subroutine stack is at 7777, and the address of the current top level of this stack decreases as the stack is filled.)

		Just Before CALL SUB1(A)	Just After CALL SUB1(A)	Just After SETGRD G2 Instruction	Just After Return from SUB2	Just After Return from SUB1
Outer program	7777	0	0	0	0	0
Return Address	7776	0	0015	0015	0015	0
Group Name Ptr.	7775	7770	7770	7770	7770	7770
Stack Level	7774	0	7000	7000	7000	0
Current Grid Ptr.	7773	7761	7761	7761	7761	7761
Group Name (MAIN)	7772	4040	4040	4040	4040	4040
	7771	1116	1116	1116	1116	1116
	7770	1501	1501	1501	1501	1501
Current Grid	7767	36	36	36	36	36
	7766	24	24	24	24	24
	7765	2	2	2	2	2
	7764	144	144	144	144	144
	7763	62	62	62	62	62
	7762	0	0	0	0	0
	7761	3	3	3	3	3
SUB1 Outer program	7760		7777	7777	7777	
SUB1 Return Address	7757		0	0027	0	
SUB1 Group Name Ptr.	7756		7770	7770	7770	
SUB1 Stack Level	7755		0	7002	0	
SUB1 Current Grid Ptr.	7754		7761	7761	7761	
SUB2 Outer program	7753			7760		
SUB2 Return Address	7752			0		
SUB2 Group Name Ptr.	7751			7744		
SUB2 Stack Level	7750			0		
SUB2 Current Grid Ptr.	7747			7736		
SUB2 Group Name (EXT)	7746			4040		
	7745			2440		
	7744			0530		
SUB2 Current Grid	7743			144		
	7742			0		
	7741			2		
	7740			113		
	7737			62		
	7736			2		

Contents of Subroutine Stack During Program in Figure 18

FIGURE 21

Form AEC-427
(6/72)
AECM 3201

U.S. ATOMIC ENERGY COMMISSION

UNIVERSITY–TYPE CONTRACTOR'S RECOMMENDATION FOR DISPOSITION OF SCIENTIFIC AND TECHNICAL DOCUMENT

(See Instructions on Reverse Side)

AEC REPORT NO.	2. TITLE
COO-2383-0019	A GRAPHICS INTERPRETER LANGUAGE

TYPE OF DOCUMENT (Check one):

☒ a. Scientific and technical report
☐ b. Conference paper
Title of conference _____
Date of conference _____
Exact location of conference _____
Sponsoring organization _____
☐ c. JOURNAL ARTICLE Submitted to _____
☐ d. Other (Specify)

RECOMMENDED ANNOUNCEMENT AND DISTRIBUTION (Check one):

☒ a. AEC's normal announcement and distribution procedures may be followed.
☐ b. Make available only within AEC and to AEC contractors and other U.S. Government agencies and their contractors.

REASON FOR RECOMMENDED RESTRICTIONS:

SUBMITTED BY: NAME AND POSITION (Please print or type)

C. W. Gear
Professor & Principal Investigator

Organization

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Signature	Date
C. W. Gear	May 8, 1975

FOR AEC USE ONLY

AEC CONTRACT ADMINISTRATOR'S COMMENTS, IF ANY, ON ABOVE ANNOUNCEMENT AND DISTRIBUTION RECOMMENDATION:

PATENT CLEARANCE:

☐ a. AEC patent clearance has been granted by responsible AEC patent group.
☐ b. Report has been sent to responsible AEC patent group for clearance.
☐ c. Patent clearance not required.

BLIOGRAPHIC DATA EET	1. Report No. UIUCDCS-R-75-710	2.	3. Recipient's Accession No.

Title and Subtitle

A GRAPHICS INTERPRETER LANGUAGE

5. Report Date: May, 1975

6.

Author(s)

JAMES BRENDAN STYNES

Performing Organization Name and Address

Department of Comp er Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

8. Performing Organization Rept. No. UIUCDCS-R-75-710

10. Project/Task/Work Unit No.

11. Contract/Grant No.

US AEC AT(11-1) 2383

Sponsoring Organization Name and Address

US AEC Chicago Operations Office
9800 South Cass Avenue
Argonne, Illinois

13. Type of Report & Period Covered

14.

Supplementary Notes

Abstracts

A "high level" machine language for efficient implementation of a multiterminal graphics system support package is described. The type of support package for which this is intended is an interactive network design system using relatively static displays, but making much use of user defined and system menus for command and control. The language is executed via an interpreter on a PDP-8 with disk storage.

Key Words and Document Analysis. 17a. Descriptors

Graphics
Interactive
Machine language
Time sharing
Network analysis

Identifiers/Open-Ended Terms

COSATI Field/Group

Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 140
Unlimited	20. Security Class (This Page UNCLASSIFIED	22. Price